Constraint Propagation in Mozart

Tobias Müller

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Saarbrücken, 2001

Programming Systems Lab
Universität des Saarlandes, 66041 Saarbrücken, Germany
Email: `tmueller@ps.uni-sb.de`
Web: `www.ps.uni-sb.de/~tmueller`

This document was prepared with LaTeX 2_ε.
Program listings were typeset with a modified version of Denys Duchier's `raw2tex`.

Prüfungsausschuss / Examining Committee:

Vorsitzender:	Prof. Dr. Jörg Siekmann
Erstgutachter:	Prof. Dr. Gert Smolka
Zweitgutachter:	Assistant Prof. Dr. Martin Henz
Dekan:	Prof. Dr. Rainer Schulze-Pillot
Beisitzer:	Dr.-Ing. Alassane Ndiaye
Tag des Kolloquiums:	3. Dezember 2001

Abstract

This thesis presents constraint propagation in Mozart which is based on computational agents called propagators. The thesis designs, implements, and evaluates propagator-based propagation engines. A propagation engine is split up in generic propagation services and domain specific domain solvers which are connected by a constraint programming interface. Propagators use filters to perform constraint propagation. The interface isolates filters from propagators such that they can be shared among various systems.

This thesis presents the design and implementation of a finite integer set domain solver for Mozart which reasons over bound and cardinality approximations of sets. The solver cooperates with a finite domain solver to improve its propagation and expressiveness.

This thesis promotes constraints to first-class citizens and thus, provides extra control over constraints. Novel programming techniques taking advantage of the first-class status of constraints are developed and illustrated.

Kurzzusammenfassung

Diese Dissertation beschreibt Constraint-Propagierung in Mozart, die auf Berechnungsagenten, Propagierer genannt, basiert. Die Dissertation entwirft, implementiert und evaluiert Propagierer-basierte Propagierungsmaschinen. Eine Propagierungsmaschine ist aufgeteilt in generische Propagierungsdienste und domänenspezifische Domänenlöser, die durch eine Schnittstelle zur Constraint-Programmierung miteinander verbunden sind. Propagierer benutzen Filter, um Constraints zu propagieren. Die Schnittstelle isoliert Filter von Propagierern, so dass Programmkodes von Filtern von verschiedenen Systemen genutzt werden können.

Diese Dissertation präsentiert den Entwurf und die Implementierung eines Domänenlösers über endliche Mengen von ganzen Zahlen für Mozart, die über Mengen- und Kardinalitätsschranken approximiert werden. Dieser kooperiert mit einem Löser über endlichen Bereichen, um die Propagierung und die Ausdrucksfähigkeit zu verbessern.

Diese Dissertation erhebt Constraints zu emanzipierten Datenstrukturen und stellt auf diese Weise zusätzliche Steuerungsmöglichkeiten über Constraints zur Verfügung. Des Weiteren werden neuartige Programmiertechniken für emanzipierte Constraints entwickelt und demonstriert.

Extended Abstract

This thesis presents constraint propagation in Mozart, a programming system for the concurrent constraint programming language Oz. The presented framework of constraint propagation is based on propagators which are concurrent computational agents realizing constraints.

Constraint Propagation Engines This thesis designs an architecture for propagation engines which perform propagator-based constraint propagation. A propagation engine comprises a generic part, called propagation services, and a domain-specific part, called domain solver. A domain solver can be regarded as an abstract data type encapsulating all domain specific details. Thus, the propagation services can be generic by relying on the abstractions of the abstract data type. The integration of propagation services into the virtual machine of Mozart is presented and an interface between propagation services and domain solvers for constructing plug-in domain solvers is developed.

A propagator uses a filter to perform constraint propagation. The interface is designed such that a filter can be shared among various systems by isolating the filter from the actual propagator.

The implementations of both the propagation services and the interface are presented in detail. Further, the implementation of a finite domain solver is given to demonstrate the presented framework.

Propagation services and the interface have been implemented as part of the virtual machine of Mozart while the domain solvers are implemented as plug-in domain solvers. The efficiency of the obtained constraint engines is competitive with existing systems.

Finite Integer Set Constraints This thesis presents the design and implementation of a full-fledged domain solver over finite sets of integers for Mozart. This solver reasons over bound approximations as well as over the cardinalities of sets. The solver is tightly connected with the finite domain solver of Mozart to improve expressiveness and constraint propagation. Resulting programming techniques are demonstrated by various case-studies.

The thesis presents a scheme for generating filters for a set propagators from the corresponding set expressions. The finite integer set domain solver is used for applications in computational linguistics and combinatorial optimization. The implementation of the finite set solver is based on the presented framework of propagation engines.

First-class Constraints This thesis promotes constraints to first-class citizens. Constraint propagation of existing propagation-based solvers is restricted to the values of the variables connected to the constraints. Promoting constraints to first-class citizens gives the programmer an additional level of control over constraints and makes it possible to go beyond the propagation of values.

First-class constraints make it possible to supplement conventional constraint propagation by symbolic constraint reasoning. This makes novel programming techniques possible. These techniques can improve existing solvers and make it possible to enter new application areas by constraint programming. Several case-studies demonstrate the benefits of constraint programming with first-class constraints.

First-class constraints are prototypically integrated in Mozart. The integration requires only conservative extensions of the propagation engine framework.

Debugging the correctness of propagation-based constraint engines may require to investigate the state of a solver at certain stages. This thesis presents a debugging scheme which is based on the investigation of solver states by different graph-views. Further, a debugging tool is derived from the proposed scheme, called Investigator. Its use is illustrated in an example debugging session. The implementation of the Investigator is fully based on first-class constraints.

Ausführliche Zusammenfassung

Diese Dissertation beschreibt Constraint-Propagierung in Mozart, ein Programmiersystem für die nebenläufige Constraint-Programmiersprache Oz. Das beschriebene Schema von Constraint-Propagierung basiert auf Propagierern, die als nebenläufige Berechnungsagenten Constraints realisieren.

Constraint-Propagierungsmaschinen Diese Dissertation entwirft eine Architektur für Propagierungsmaschinen, die Propagierer-basiert Constraints propagieren. Eine Propagierungsmaschine besteht aus einem generischen Teil, genannt Propagierungsdienst, und einem domänenspezifischen Teil, genannt Domänenlöser. Ein Domänenlöser kann als abstrakter Datentyp angesehen werden, der alle domänenspezifischen Details einschließt. Auf diese Weise können Propagierungsdienste generisch auf domänenspezifische Details über Abstraktionen des abstrakten Datentyps zugreifen. Die Integration von Propagierungsdiensten in die virtuelle Maschine von Mozart wird beschrieben und eine Schnittstelle zwischen Propagierungsdiensten und Domänenlösern zum Konstruieren von Plug-in-Lösern entwickelt.

Ein Propagierer benutzt einen Filter, um Constraints zu propagieren. Die Schnittstelle wird durch Isolieren von Filtern und Propagierern so entworfen, dass der Programmkode eines Filters von verschiedenen Systemen genutzt werden kann.

Die Implementierung von Propagierungsdiensten und der Schnittstelle zur Constraint-Programmierung wird im Detail beschrieben. Zur Illustration wird die Implementierung eines Domänenlösers über endlichen Domänen erläutert.

Propagierungsdienste und die Schnittstelle werden als Teil der virtuellen Maschine implementiert, während Domänenlöser als Plug-in-Löser realisiert werden. Die erzielte Effizienz der implementierten Constraint-Maschinen ist konkurrenzfähig mit existierenden Systemen.

Constraints über endlichen Mengen von ganzen Zahlen Diese Dissertation beschreibt den Entwurf und die Implementierung eines vollausgestatteten Domänenlösers über endlichen Mengen von ganzen Zahlen für Mozart. Dieser Löser propagiert über Mengen- und Kardinalitätsschranken und ist darüber hinaus eng mit dem Löser über endlichen Bereichen von Mozart verbunden, um die Ausdrucksfähigkeit und die Propagierung zu verbessern. Fallstudien illustrieren resultierende Programmiertechniken.

Des Weiteren beschreibt diese Dissertation ein Schema zum Erzeugen von Filtern für Mengenpropagierer aus korrespondierenden Mengenausdrücken. Der Constraint-Löser über endlichen Mengen von ganzen Zahlen wird für Anwendungen in der Computerlinguistik und der kombinatorischen Optimierung eingesetzt. Die Implementierung des Lösers basiert auf dem Schema von Propagierungmaschinen.

Emanzipierte Constraints Diese Dissertation erhebt Constraints zu emanzipierten Datenstrukturen. Das Lösen von Constraints in existierenden propagierungsbasierten Lösern ist auf die Werte der mit den Constraints verbundenen Variablen beschränkt. Das Erheben von Constraints zu emanzipierten Datenstrukturen stellt dem Programmierer eine zusätzliche Ebene der Steuerung über Constraints zur Verfügung und ermöglicht es, über das Propagieren von Werten hinauszugehen.

Emanzipierte Constraints ermöglichen es, herkömmliche Constraint-Propagierung um symbolisches Constraint-Lösen zu erweitern. Das macht neuartige Programmiertechniken möglich, die existierende Löser verbessern und neue Anwendungsgebiete für Constraint-Programmierung erschließen können. Mehrere Fallstudien demonstrieren die Vorzüge von Constraint-Programmierung mit emanzipierten Constraints.

Emanzipierte Constraints sind prototypisch in Mozart integriert, und deren Integration erfordert nur konservative Erweiterungen des Schemas von Propagierungsmaschinen.

Die Fehlersuche zur Korrektheit von propagierungsbasierten Constraint-Lösern kann das Untersuchen des Zustandes eines solchen in einer bestimmten Lösungsphase erfordern. Diese Dissertation beschreibt ein Fehlersuchschema, das auf dem Untersuchen des Zustandes eines Lösers durch verschiedene Graph-basierte Darstellungen beruht. Des Weiteren wird ein Werkzeug zur Fehlersuche abgeleitet, Investigator genannt. Sein Gebrauch wird am Beispiel einer Fehlersuche demonstriert. Die Implementierung des Investigators basiert vollkommen auf emanzipierten Constraints.

Acknowledgements

I am grateful to Gert Smolka for giving me the opportunity to work with the Programming Systems Lab in Saarbrücken and for teaching me that striving for simplicity and clarity is the key to gain novel insights. It was a privilege and fun to work in his group and I thank all colleagues for a stimulating atmosphere.

Discussions with Christian Schulte, Martin Henz and Jörg Würtz about design and implementation of constraint solving were an invaluable source of ideas and inspirations. Martin Henz gave me the opportunity to cooperate with him and other researchers in the FIGARO project.

It was a pleasure to work with Konstantin Popov, Ralf Scheidhauer, Michael Mehl and Christian Schulte on the implementation of Mozart. I profited a lot from their experience and knowledge.

Discussions with Carmen Gervet helped me to find a starting point for my work on finite set constraints.

I am grateful to Martin Müller and Denys Duchier for sharing with me their knowledge on set constraints. Denys Duchier helped to improve the implementation a lot by using set constraint in computer linguistic applications and by contributing ideas for the scheme for generating filters.

Christian Schulte, Thorsten Brunklaus, Katrin Erk, Sven Thiel, Leif Kornstaedt, Konstantin Popov, and Denys Duchier gave me invaluable feedback on draft versions of this thesis and thus, helped me to improve the presentation of my work. Of course, remaining errors are my fault.

Marco Kuhlmann helped me to master LaTeX and all the rest of it. Mats Carlsson and Joachim Schimpf helped me programming the benchmarks in Chapter 10 for SICSTUS and *ECL^{i}PS^e*, respectively.

I thank Erica Melis and Jürgen Zimmer for giving my the opportunity to apply first-class constraints to proof planning problems.

I will keep all of the above mentioned in good memory; some of them became good friends of mine.

Last but not least, I thank Simone for her patience, support and love. She is and will always be a very special person to me.

To Arno, Christel and Margarete.

Contents

Chapter 1

Introduction

Everybody faces constraints everyday. Examples for constraints are: the number of persons being carried by a lift must not exceed 10 and the open hours of a shop are from 10am to 4pm. These constraints are easy to meet but as the number of constraints grows, a procedure for solving constraints is desired. Such a procedure is implemented by a *constraint solver*.

Constraint Problems and Constraint Solvers A *constraint problem* is any problem which is expressed in terms of problem variables, values and constraints over variables and value. The domain of the values is called *constraint domain*. For example, the number of persons being carried by a lift can be represented by a variable *persons*, the limit by the value 10 (of the domain of positive integers) and the restriction that the number of persons must not exceed the limit by the constraint $persons < 10$. A problem expressed in terms of constraints is submitted to a constraint solver which tries to assign valid values to the problem variables satisfying all constraints of the problem. Such an assignment is called a *solution* of the problem. It is often the case that a solution is required to be optimal according to a criterion leading to an *optimization problem*.

Various (specialized) constraint solvers have been devised over the years, *e.g.*, the Simplex algorithm to solve linear programming problems [35, 31] or local search for propositional satisfiability problems [58, 133, 1]. This thesis is about propagation-based constraint solving.

Propagation-based Constraint Solvers A *propagation-based constraint solver* uses two techniques to find assignments: deterministic *constraint propagation* and non-deterministic *search*. Constraint propagation excludes those values from being assigned to a variable which are incompatible with a solution. Not yet excluded values are stored in the *domain* of the variable (called *constraint variable*). Incompatible values are excluded (or filtered) from domains by *propagators*. A propagator represents an individual constraint over a number of problem variables (its *parameters*) and encapsulates

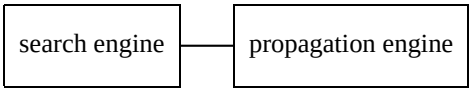


Figure 1.1: Architecture of a propagation-based constraint solver.

an algorithm for filtering out incompatible values.[1] Such an encapsulated algorithm is called *filter algorithm*.

Constraint propagation is typically *not* able to exclude all but one value per variable and thus, to produce an assignment for a given problem. Hence, constraint propagation is complemented by search. As soon as constraint propagation stopped because no further values can be excluded, search branches to different alternatives in a speculative way excluding values and thus, triggering new constraint propagation in the alternatives. Constraint propagation and search are interleaved until a desired solution is found or all alternative are explored.

Constraint propagation is performed by a *propagation engine* while search is done by a *search engine*. Both engines together form a *propagation-based constraint solver* (Figure 1.1).

Propagation Engines Constraint propagation with propagators is performed by a propagation engine (Figure 1.2). A propagation engine consists of generic *propagation services* and a constraint domain specific solver, called *domain solver*. A domain solver provides the variables and propagators for a certain constraint domain while the propagation services manage and control these variables and propagators in a domain-independent way.

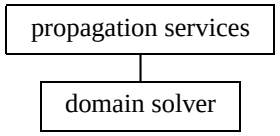


Figure 1.2: Architecture of a propagation engine.

Subject of this Thesis This thesis investigates constraint propagation with propagators in three directions: (i) it designs and implements propagation engines (Section 1.1); (ii) it designs a domain solver over finite sets of integers (Section 1.2); (iii) it promotes propagators to first-class citizens (Section 1.3).

This thesis is in a line with other theses about the design and implementation of Mozart [124, 150, 63]. In particular, this thesis is connected to Mehl's thesis on the design and implementation of the virtual machine of Mozart [88] and Schulte's thesis on design and implementation of search in Mozart [128].

1.1 Constraint Propagation Engines

Part I of this thesis presents the integration of constraint engines in Mozart [99].

Motivation Mozart is a programming system implementing the programming language Oz which provides for speculative computation by computation spaces and concurrency and synchronization by threads and logic variables [137]. Threads and computation spaces form a unique environment for propagation engines in contrast to todays constraint solvers. These are mostly Prolog-based (*e.g.*, GNU PROLOG [38], SICSTUS [74], *ECL^{i}PS^e* [76] and CHIP [40]) or C++ libraries (*e.g.*, ILOG SOLVER [73]). Further, the key requirements for propagation engines are (i) to be able to use sophisticated filter algorithms, (ii) to be extendible by new plug-in domain solvers and (iii) to be efficient.

[1] The notion *filtering* occurred first in [148]. Another notion for filtering found in the literature is *narrowing*.

Architecture An architecture for propagation engines is developed which is based on propagators and constraint variables. This architecture separates constraint domain-independent propagation services from constraint domain-dependent domain solvers. The separation is achieved by splitting propagators and constraint variables in their domain independent parts, their heads, and in their domain dependent parts, their bodies. Thus, the separation in propagation services and domain solvers comes naturally. The architecture is refined by defining an interface between propagation services and domain solvers. This interface is called *constraint propagator interface* (for short CPI). This makes it possible to plug domain solvers as external modules into propagation services provided by some runtime system.

A propagator computes the values to be excluded from its parameters by running a filter. Most of the effort for implementing a new domain solver is needed for designing and implementing the filter of the propagators. Hence it is desired that filters are shared between various domain solver implementations. Consequently, an interface between propagators and filter (called *filter interface*) is conceived making the exchange of filter implementation between different constraint solvers possible and straightforward.

Integration This thesis develops a model for integrating propagation services and an interface to domain solvers into the virtual machine of Mozart. This model extends the virtual machine of Mozart in an orthogonal and conservative way since it does not change the virtual machine; it only uses parts of the virtual machine via a clean interface or adds new functionality without changing old one.

Implementation The implementation of propagation engines follows the integration model and is provided by Mozart in production quality. The implementation covers three parts: propagation services as part of the virtual machine, the interface for connecting domain solvers, and the implementation of a complete plug-in finite domain solver (including constraint variables and a propagator for the constraint $x \leq y + c$). The presentation is supplemented by code samples to support a reconstruction and the plug-in finite domain solver can be obtained from [105].

Evaluation The plain propagation performance is measured for Mozart and other propagation-based constraint solvers. This is done by running an inconsistent finite domain constraint in various configurations and taking the time until an inconsistency is detected. The results of the measurements obtained for Mozart are compared with results for the other constraint solvers.

A central feature of the design and implementation of propagation engines is factorization by introducing interfaces. Hence, the performance impact of various interfaces is measured and analyzed in depth to reveal the computational cost of this factorization.

1.2 Finite Integer Set Constraints

Part II of this thesis designs a domain solver over finite set of integers which is integrated and demonstrated in Mozart.

Motivation Sets occur in many problems as a natural means of expression. Those sets

are subsets of some finite universe and their elements can be mapped against positive integers without any loss of expressiveness. Hence, a domain solver over *finite integer set constraints* is integrated in Mozart. This solver is particularly useful for applications in computational linguistics [48, 80, 45].

Constraints over Finite Integer Sets This solver approximates a set by a lower bound set and an upper bound set and additionally, by cardinality bounds. The lower bound set contains elements known to be in the set. The upper bound set contains elements which are still candidates to be in the set. Constraint propagation adds elements to the lower bound and removes elements from the upper bound. The approximation by lower and upper bounds was inspired by Gervet's work on set intervals [55].

Problem specifications provide frequently hints about the number of elements to be expected in sets. This is taken into account by adding cardinality bounds. Cardinality bounds denotes the minimal and maximal number of elements of a set. Cardinality propagation raises the minimal number of elements and lowers the maximal number of elements. This makes it possible to determine sets earlier or to detect inconsistencies faster.

Generation of Filters The design of filters performing bound and cardinality propagation is not straightforward and prone to errors. As a consequence, this thesis proposes a scheme for automatically generating filter for set constraints. The idea is to express a set constraint in terms of constraints with primitive filters and to generate a monolithic filter for the set constraint by transformations on the primitive filters. The implementation of the devised scheme generates filters for bounds and cardinality propagation. These filters can be directly used by set propagators via the filter interface.

Integration and Application Finite integer set constraints are integrated into Mozart as domain solver which a constraint program accesses by a library. The use of set constraints in Mozart is illustrated by applications in the field of combinatorial optimization and natural language processing. Particularly, the combination of finite sets and finite domains is essential to improve expressiveness and constraint propagation and to make the search for optimal solutions possible.

1.3 First-class Constraints

Part III of this thesis promotes constraints to first-class citizens.

Motivation Traditional constraint solver regard individual constraints as anonymous entities. The only way to gain control over a constraint is to connect it with a $0/1$-variable and to reflect the constraint's validity to this variable. Such a constraint is called *meta* or *reified* constraint [143, 112]. The control over a reified constraint is very limited since it eventually imposes a constraint. As consequence, constraints are promoted to first-class citizens and thus, full control over constraints is gained and *true* meta constraint programming is made possible.

Promoting Constraints to First-class Citizens Constraints are promoted to first-class citizens by providing an abstract data type for constraints. Values of this type can occur

at the same place where primitive values can occur. The operations of the abstract data type make it possible to obtain the name and the parameters of a first-class constraint, to learn whether the constraint is already entailed or not, to explicitly discard a first-class constraint and to turn its propagation on or off.

Programming with First-class Constraints In contrast to constraint propagation, which excludes values from variables, first-class constraint programming reasons about the constraints themselves by accessing the current state of a constraint and controlling a constraint's behavior directly. This makes new programming and inference techniques possible. Promising applications in the field of combinatorial optimization are demonstrated. These fields include early detection of unsatisfiable constraints, re-formulation of constraints to improve propagation, and garbage collection of redundant but not yet entailed constraints.

Implementation First-class constraints have been prototypically implemented with Mozart. The implementation is orthogonal to the propagation services and does not need any modification of domain solvers. Further, no performance penalty is imposed when not using first-class constraints.

Debugging Support for Propagation Engines This thesis develops a scheme for debugging the correctness of a constraint program which is in contrast to current schemes which focus on optimizing search and performance [134, 126, 91]. The proposed scheme investigates the state of the corresponding propagation engine which can be regarded as a graph. The nodes and edges of the graph are derived from the relations between propagators and their parameters. Such a graph is visualized by various graphs views. The proposed debugging scheme is realized in Mozart by a debugging tool called *Constraint Investigator* which makes it possible to explore graph views derived from a state of a propagation engine. The implementation of the Investigator is based on first-class constraints.

1.4 Published Material

The results of this thesis have been partly published in the articles listed below.

Constraint Propagation Engines The constraint propagator interface has been presented in:

> Tobias Müller and Jörg Würtz. Embedding propagators in a concurrent constraint language. *The Journal of Functional and Logic Programming*, 1999 [108].

which is an extended version of [107]. An interface to separate the implementation of a propagator from its filter algorithm has been proposed in:

> Ka Boon Ng, Chiu Wo Choi, Martin Henz, and Tobias Müller. GIFT: a generic interface for reusing filtering algorithms. *Proceedings of the Workshop on Techniques for Implementing Constraint Programming Systems - TRICS*, 2000 [111].

Finite Integer Set Constraints The finite integer set solver of Mozart and corresponding programming techniques have been presented in:

> Tobias Müller and Martin Müller. Finite set constraints in Oz. *13. Workshop Logische Programmierung*, 1997 [106] .

First-class Constraints Constraints as first-class citizens have been presented in:

> Tobias Müller. Promoting constraints to first-class status. *Proceedings of the First International Conference on Computational Logic – CL2000*, July 2000 [102].

A constraint debugger for constraint solving with propagators has been presented in:

> Tobias Müller. Practical investigation of constraints with graph views. *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming – CP 2000*, 2000 [101].

1.5 Overview

Chapter 2 introduces constraint programming and presents a computation model for constraint propagation. Chapter 3 introduces constraint programming in Oz.

The material of this thesis is presented in three parts.

Constraint Propagation Engines (Part I) Chapter 4 develops an architecture for propagator-based constraint engines. Chapter 5 discusses the integration of propagation services into the virtual machine of Mozart. Chapter 6 develops an interface for connecting domain solvers with propagation engines. Chapter 7 discusses the implementation aspects of the integration of propagation services into the virtual machine of Mozart. Chapter 8 presents the constraint propagator interface of Mozart as an concrete interface for implementing domain solvers and illustrates its application by implementing a complete plug-in finite domain solver. Chapter 9 discusses the integration of aliasing of constraint variables in constraint propagation engines. Chapter 10 evaluates the obtained propagation engines in detail and compares Mozart with other available constraint solvers.

Finite Integer Set Constraints (Part II) Chapter 11 presents finite integer set constraints. Chapter 12 develops a scheme for generating filter algorithms for finite set propagators and illustrates the scheme by an example. Chapter 13 presents the finite set library for Mozart and demonstrates its use by various case studies.

First-class Constraints (Part III) Chapter 14 introduces constraints as first-class citizens and develops programming techniques for first-class constraints. Chapter 15 presents a constraint debugging tool which uses a graph metaphor to visualize the state of a solver.

Chapter 16 summarizes the contributions of this thesis and gives an outlook to future work. Supplementary material to the thesis can be found at [105].

Chapter 2

Problem Solving with Constraints

This chapter introduces informally constraint solving as combination of propagation, branching and exploration (Section 2.1). Then, constraint satisfaction problems as a formal model for constraint solving are presented (Section 2.2). Finally, a computational model for constraint solving with propagators is given which is the foundation for designing and implementing propagation engines (Section 2.3).

2.1 Constraint Solving

Constraint solving is a method to solve combinatorial optimization problems. A problem is expressed by a set of constraints and a set of variables, called *problem variables*. The *constraints* state relations between variables. A possible *solution* is an assignment of values to variables satisfying all constraints. To eventually compute an assignment for a problem variable, such a problem variable is annotated with a set of potential values, called its *domain*. A variable denotes a value if its domain contains only a single value. Such a variable is *determined*. A variable is determined by *constraint propagation*, by successively removing values from a variable's domain (short *variable domain*) which are incompatible with the constraints imposed on this variable. Constraint propagation is implemented by a *propagation algorithm*.

Constraints are expressed with a constraint language which defines the constraint domains and the available constraints for the domains. Common constraint domains are finite domain constraints [145], finite set constraints [55], real interval constraints [12] and record constraints [138, 123]. The way a problem is expressed by constraints controls the propagation behavior obtained by the constraint solver. Since constraint propagation on its own usually not able to find a solution, it is supplemented by search. Search tries to solve the problem under an assumption and if this assumption is wrong, another assumption is tried on the same problem. This process is recursively continued by alternating constraint propagation and search until a solution is obtained. Thus, a *search tree* is created. This approach has two dimensions: (i) the shape (or branching) of the search tree and (ii) the way the search tree is explored. Dimension (i) is implemented by a *branching algorithm* and dimension (ii) by a *exploration algorithm*.

Propagation, branching and exploration algorithms are implemented by a *constraint program* which is submitted to a constraint solver.

Propagation Algorithm A propagation algorithm enforces a relation on the values of problem variables by imposing constraints on the problem variables. Once imposed, a constraint enforces the relation it represents by removing those values from the domains which are not in the relation. Moreover, a constraint reconsiders the values of its variables as soon as values are removed by other constraints. The purpose of a propagation algorithm is to remove values from variable domains *resp.* to indicate that there is no valid assignment, *i.e.* propagation fails. Both outcomes of propagation have an impact on the number of alternatives to be explored by search: The number of values left in the domains of the problem variables determines the number of alternatives to be explored. Failing propagation terminates exploration of the search tree at this point. Additionally, the propagation algorithm shall run with modest computational cost.

Branching Algorithm The branching algorithm defines the shape of the search tree. As soon as the propagation algorithm has finished and has not failed, the branching algorithm computes a branching constraint which determines what alternatives are explored. A branching constraint is a disjunction expressing alternatives. It has to be such that no solutions are lost. For example, having a problem variable $x \in \{1, 2, 3, 4\}$, possible branching constraints are: (i) $x = 1 \vee x \neq 1$, (ii) $x \leq 2 \vee x > 2$ and (iii) $x = 1 \vee x = 2 \vee x = 3 \vee x = 4$. The creation of alternatives by a branching algorithm creates a search tree where the alternatives are represented by the choice nodes and the constraints imposed on the branches are the edges of the tree.

Branching is also called *distribution* and the computation of a branching constraint is done by a *distribution step*.

Exploration Algorithm The exploration algorithm determines how a search tree created by a branching algorithm is traversed by determining the order of how the alternatives are explored (*e.g.*, in a depth-first or a breadth-first fashion).

2.2 Constraint Satisfaction and Filtering

The propagation algorithm introduced in Section 2.1 can be formalized by a *constraint satisfaction problem* (CSP) [85, 144, 95, 141, 7]. This section introduces CSPs formally and provides an algorithm for solving them.

Constraint Satisfaction Problems A CSP is a triple (V, D, C) where V is a finite set of variables $\{x_1, \ldots, x_n\}$, D is a set of mappings from variables to domains $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$ (called *domain store*) and C a finite set of constraints $\{c_1, \ldots, c_m\}$. A domain is a finite set of values and the domain d_i of a variable x_i can by retrieved by $dom(D, x_i) = d_i$ if $x_i \mapsto d_i \in D$. Variables are denoted by x and y while values by v and w. An n-tupel $(x_1 = v_1, \ldots, x_n = v_n)$ denotes the simultaneous assignment of the values $v_1, \ldots, v_n$ to the variables $x_1, \ldots, x_n$, respectively. A constraint $c(x_1, \ldots, x_m)$ with $m \leq n$ is a set of m-tupels $\{(x_1 = v_{1,1}, \ldots, x_m = v_{m,1}), \ldots, (x_l = v_{1,l}, \ldots, x_m = v_{m,l})\}$ denoting simultaneous assignments legal for constraint c. The set of parameters

$\{x_1, \ldots, x_k\}$ of a constraint $c(x_1, \ldots, x_k)$ is denoted by $V(c)$.

The set D^α of tuples denoting all possible simultaneous assignments of a given domain store D is

$$D^\alpha = \{(x_1 = v_1, \ldots, x_k = v_k) \mid v_1 \in dom(D, x_1) \wedge \ldots \wedge v_k \in dom(D, x_k)\} \quad (2.1)$$

A domain store D_1 is *stronger* than a store D_2 (*resp.* D_2 is *weaker* than D_1) if $D_1^\alpha \subseteq D_2^\alpha$. Removing values from domains is called *domain pruning* or *domain reduction*.

Domain Filtering Domain filtering of a constraint c removes for $x_i \in V(c)$ all values from the domains $dom(D, x_i)$ which are not legal with c and evolves D to D':

$$D' = filter(D, c) \text{ where } \forall x_i \in V : \quad (2.2)$$
$$dom(D', x_i) = \begin{cases} \{v_i \mid (\ldots, x_i = v_i, \ldots) \in c \wedge v_i \in dom(D, x_i)\} & \text{if } x_i \in V(c) \\ dom(D, x_i) & \text{else} \end{cases}$$

Entailment and Failure A tuple $s = (y_1 = w_1, \ldots, y_k = w_k)$ over all the variables $y_i \in V'$ and $V' \subseteq V$ is a projection of a tuple $t = (x_1 = v_1, \ldots, x_n = v_n)$ over all variables $x_i \in V$ if $y_1 = w_1, \ldots, y_k = w_k \in \{x_1 = v_1, \ldots, x_n = v_n\}$. The predicate $projection(t, s)$ is true if s is a projection of t. The set of tuples $c^\alpha = \{t \mid projection(t, s) \wedge s \in c\}$ denotes all legal simultaneous assignment tuples of c in D^α. A constraint c is *entailed* by a domain store D if

$$D^\alpha \subseteq c^\alpha \quad (2.3)$$

which means that all simultaneous assignments in D are subsumed by the simultaneous assignments of c.

A CSP *fails* if $\exists x_i \in V : dom(D', x_i) = \emptyset$. This is equivalent to $D^\alpha = \emptyset$. A failed CSP is denoted by $(V, \bot, C)$. A CSP is determined if D^α is a singleton set.

The negation of a constraint $c(x_1, \ldots, x_k)$ is $\neg c(x_1, \ldots, x_k) = \{t \mid t = (x_1 \mapsto v_1, \ldots, x_k \mapsto v_k) \wedge t \notin c\}$.

A Procedure for Constraint Propagation Constraint propagation enforces a certain degree of consistency between the values in the domains and the constraints. A common degree of consistency is *arc-consistency* which requires that every value in a domain is part of a legal simultaneous assignment of a constraint (see [141, Section 3.2.3] for a formal definition). Various consistency algorithms to achieve arc-consistency have been proposed (from AC-3 [85] and AC-5 [144] to AC-7 [15]). The algorithm in Program 2.1 is an AC-3 algorithm (see [85] and [141, Section 4.2.3]) generalized for n-ary constraints.

The algorithm in Program 2.1 imposes arc-consistency on a CSP (V, D, C) with n-ary constraints and returns an updated CSP. The algorithm represents a constraint c by a *filter function* $\mathsf{F}_c : \mathcal{D} \to \mathcal{D} \times \{0, 1\}$ which performs domain filtering *and* indicates if c is entailed or not. A filter F_c is the conjunction of an entailment function $\mathsf{E}_c : \mathcal{D} \to \{0, 1\}$ and a narrowing function $\mathsf{N}_c : \mathcal{D} \to \mathcal{D}$ where $\mathcal{D}$ denotes a set of domain stores. The entailment function $\mathsf{E}_c(D)$ returns 1 if c is entailed by D according to condition (2.3).

```
CSP : propagate(CSP : (V, D, C)) {
  S = C                                          // initialize iteration set
  while (S ≠ ∅) {                               // iterate until fixed-point
    c ∈ S; S = S \ {c}                                  // pick a constraint
    (D', entailed) = F_c(D)                          // run filter function
    if (D' = ⊥)                                       // detect failure ...
      return (V, ⊥, C)                            // .. and return failed CSP
    if (entailed = 1)                                 // detect entailment ...
      C = C \ {c}                          // ... and drop entailed constraint
                        // add dependent constraints to iteration set
    S = S ∪ {c' | c' ∈ C \ {c}, x ∈ V(c) : dom(D, x) ≠ dom(D', x) ∧ x ∈ V_c'}
    D = D'                                              // update domain store
  }
  return (V, D, C)                                       // return updated CSP
}
```

Program 2.1: A generalized AC-3 consistency algorithm for n-ary constraints based on a filter function.

The narrowing function $\mathsf{N}_c(D)$ performs filtering on D according to equation (2.2) and returns an updated domain store D'.

The algorithm performs fixed-point iteration until filtering cannot remove further values from any domain. This is achieved by maintaining an iteration set S of constraints to be considered and by adding those constraints to S whose parameters' domains have been pruned. A fixed-point is represented by the domains in D and is reached if S is empty. The algorithm terminates since the domains are finite and S runs empty if no further domain pruning is possible. As an optimization, the presented algorithm discards entailed constraints which is desirable for an efficient implementation.

Required Properties of Narrowing and Entailment Functions Würtz defines in [150, Section 2.4] properties of narrowing functions N_c and entailment functions E_c that ensure that they realize a given constrain c. The conditions are given in Figure 2.1 as relations of sets of simultaneous assignments.

Property (correct.1) states that no solution of $d \wedge c$ is discarded by a narrowing function. Property (extensional.1) states that N_c only adds information to the domain store by removing simultaneous assignments. Property (adequate.1) states that failures is detected at latest if all parameters are determined. Property (monotonic.1) states that narrowing is monotonic, *i.e.*, the result of a narrowing function is at least strong as for a weaker domain store. Monotonicity of filters is needed for computing unique propagation fixed-points independent of the execution order of filters (see [150, Section 2.5]). Property (idempotent.1) guarantees that N_c has to be applied only once on a given domain store. This property is only desired for an efficient implementation.

Property (correct.2) says that entailment is correctly detected (see condition (2.3)). Property (adequate.2) makes sure the entailment is detected at latest if a all domains are determined. Finally, property (monotonic.2) states that if entailment is detected for a

$$D^\alpha \cap c^\alpha \subseteq \mathsf{N}_c(D)^\alpha \quad \text{(correct.1)}$$
$$\mathsf{N}_c(D)^\alpha \subseteq D^\alpha \quad \text{(extensional.1)}$$
$$\text{if } D \text{ is determined and } D^\alpha \subseteq \neg c^\alpha \text{ then } \mathsf{N}_c(D) = \bot \quad \text{(adequate.1)}$$
$$\text{if } D_1^\alpha \subseteq D_2^\alpha \text{ then } \mathsf{N}_c(D_1)^\alpha \subseteq \mathsf{N}_c(D_2)^\alpha \quad \text{(monotonic.1)}$$
$$\mathsf{N}_c(\mathsf{N}_c(D)) = \mathsf{N}_c(D) \quad \text{(idempotent.1)}$$
$$\text{if } \mathsf{E}_c(D) = 1 \text{ then } D^\alpha \subseteq c^\alpha \quad \text{(correct.2)}$$
$$\text{if } D \text{ is determined and } D^\alpha \subseteq c^\alpha \text{ then } \mathsf{E}_c(D) = 1 \quad \text{(adequate.2)}$$
$$\text{if } D_1^\alpha \subseteq D_2^\alpha \text{ then } \mathsf{E}_c(D_1) \geq \mathsf{E}_c(D_2) \quad \text{(monotonic.2)}$$

Figure 2.1: Properties of filters.

weaker domain store then it is also detected for a stronger one.

Propagation algorithms as introduced in Section 2.1 are realized by CSPs. Constraint propagation of propagation engines is based the algorithm in Program 2.1.

2.3 A Model for Constraint Solving with Propagators

This section presents a model for constraint solving based on propagators. This propagator-based model integrates the CSP model in the Oz computation model presented in [137]. Further, by refining the CSP model (for example with propagation events), the propagator-based model serves as a foundation for the design and implementation of propagation engines (Part I).

Basic Constraints A *basic constraint* denotes which values a variable can take. A *constraint store* hosts the conjunction of individual basic constraints and corresponds to the domain store in Section 2.2. Basic constraints $x \in d$ (variable x takes its value in domain d) and $x = y$ (variables x and y have the same value, *i.e.*, they are *aliased*) in a constraint store b_s are regarded.[1] A basic constraint $x = v$ is an abbreviation for $x \in \{v\}$. The domain d of a variable x corresponds to $dom(x, D)$ in Section 2.2 where D is the domain store corresponding to a constraint store b_s.

Non-basic Constraints A *non-basic constraint* expresses a relation (constraint) between variables, as for example between integers $x_1 + x_2 = x_3$ or between sets $s_1 \cup s_2 = s_3$. Such a relation cannot be expressed explicitly by the constraint store and it is realized by a filter which is encapsulated in a *propagator*. A propagator is a concurrent computational agent imposed on a set of variables, called its *parameters*. Dually, from the view of a variable, the propagators imposed on a variable are called *connected propagators* of a variable.

A propagator p_c enforces the constraint c on its parameters by running its filter F_c on its parameters . The propagator writes the domains pruned by its filter F_c to the basic

[1] Aliasing is an design decision of Oz and separately treated in Chapter 9.

constraint b_p of its parameters and thus, advances the store b_s to the store $b_s \wedge b_p$ if $b_s \wedge c$ entails b_p (see condition correct.1 in Figure 2.1) and b_p adds new and consistent information to b_s. A propagator p_c ceases to exist if c is entailed by b_s (the entailment function of F_c returns 1) or if $b_s \wedge c$ is unsatisfiable (dis-entailed, *i.e.*, $\neg c$ is entailed by b_s). A propagator detects entailment or dis-entailment at the latest if all its parameters denote single values according to conditions (adequate.2) and (adequate.1), respectively.

Typically, there are many propagators imposed on a set of variables leading to a sharing of variables among different propagators. This causes propagators to trigger each other by writing basic constraints to the variables. This goes on until a *propagation fixed-point* is reached which is the case if no further new basic constraints can be written to the variables (as reflected by Program 2.1).

Constraint Graph The relationship between the entities of the model, namely variables and propagators, can be depicted as a directed *constraint graph*. Variables are represented by *variable nodes* and propagators by *propagator nodes*. The constraint graph represents the current state of the CSP representing a given problem. Constraint propagation transforms the constraint graph from one state to another by applying filters to propagator parameters until eventually no propagator node is left. Thus, filters of propagators act as graph transformers: if a filter F_c detects entailment of c the corresponding propagator node is removed; but a filter can also create new propagators and thus, it adds new propagator nodes to the constraint graph.

As example consider the following propagators: $p_1(v_1, v_2) \wedge p_2(v_1, v_3) \wedge p_3(v_2, v_3)$. The corresponding graph is shown in Figure 2.2. An edge (p, v) denotes that variable v is a parameter of propagator p. An individual propagator observes and writes its basic constraints to all those variables which it is able to reach in the graph. For example, p_2 observes and writes basic constraints to v_1 and v_3 but not to v_2.

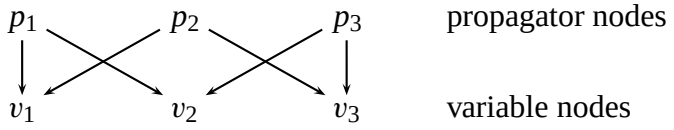


Figure 2.2: Constraint graph connecting propagators with variables.

For this constraint graph abstraction it is sufficient to assume that a propagator is an active computational agent permanently monitoring its parameters.

Events On sequential hardware, it is computationally unfeasible to have propagators permanently monitoring their parameters. Moreover, a propagator can only take advantage of certain kinds of changes to its parameters. For example, a propagator performing domain propagation benefits typically from individual values removed while a propagator performing value propagation typically exploits parameters becoming single values. Hence, different *propagation events* (for short *events*) are distinguished which define conditions for re-executing propagators. An event is denoted by e.

As an example regard finite domain constraints. One can distinguish a value event (exactly one value is left), a bounds event (bounds are narrowed) and a domain event

(some value is removed). Interestingly, events are typically (partially) ordered (forming a lattice). In case of finite domain constraints, a value event causes also bound and domain events while a bounds event causes also a domain event. Note distinguishing lower and upper bound events results in a partial order.

Constraint Graph with Events The introduction of events is reflected in the constraint graph by additional edges from variables to propagators. These additional edges are labeled with events triggering re-execution of the connected propagators. A constraint graph with events is a cyclic graph since there are edges from propagators to variables (the propagators' parameters) and from variables to propagators.

Augmenting the previous example by some events $e_{1,2,3}$ yields: $p_1(v_1^{e_1}, v_2^{e_2}) \wedge p_2(v_1^{e_2}, v_3^{e_2}) \wedge p_3(v_2^{e_2}, v_3^{e_3})$. The notation $p(v^e)$ means that p is rerun if the event e occurs on v. The corresponding extension of the constraint graph in Figure 2.2 to a constraint graph events is shown in Figure 2.3. (The complete constraint graph with events consists of the graphs in Figures 2.2 and 2.3.)

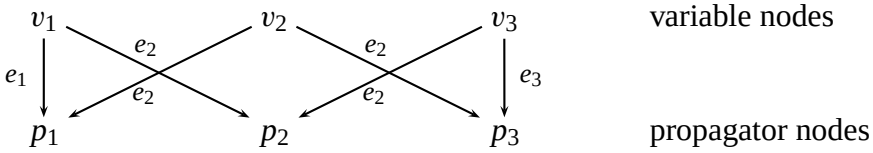


Figure 2.3: Extension of the constraint graph in Figure 2.2 by edges from variables to propagators labeled with events.

Now a propagator needs not to permanently observe its parameters for possible changes. The propagator is notified if awaited events occur. For example, p_1 is rerun if event e_1 occurs on variable v_1 or event e_2 occurs on variable v_2. Events are caused by other propagators sharing variables as parameters. For example, p_1 shares parameters with p_2 (v_1) and p_3 (v_2).

Execution States of a Propagator A propagator goes through various *execution states* during its *life-cycle* (Figure 2.4). The execution state is either `running`, `runnable`, `sleeping`, `failed`, or `entailed`. A propagator is switched from one execution state to another one according to the result of its last execution or by some external event.

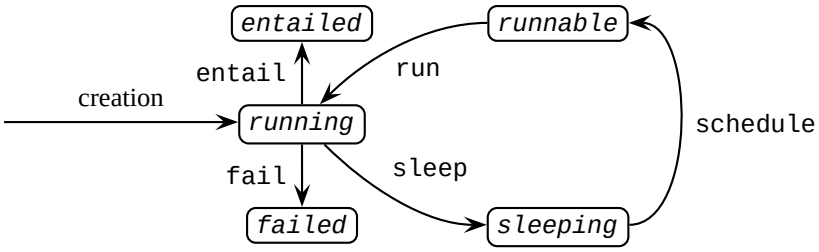


Figure 2.4: Execution states of a propagator.

On creation, the propagator p_c for constraint c becomes `running`, *i.e.*, p_c computes

a new basic constraint b_p which advances the b_s to $b_s \wedge b_p$. If $c \wedge b_s$ is not satisfiable then this is indicated by `fail` and the propagator is set to *`failed`*. In case c is entailed by b_s, this is indicated by `entail` and the propagator is set to the execution state *`entailed`*. In both cases the propagator ceases to exist.

Otherwise the propagator signals `sleep` and is set to *`sleeping`*. Propagation of other propagators may cause events on shared parameters and thus the propagator is *scheduled* (transition `schedule`) by setting its state to *`runnable`*. Once *`runnable`*, the propagator is executed by performing transition `run`. Note this model assumes the transition `run` to be *fair*, *i.e.*, every runnable propagator will eventually be run.[2] The cycle restarts and continues until no propagator causes any new event, constraint propagation has reached a fixed-point.

[2] Additionally, filters of propagators are of course assumed to terminate.

Chapter 3

Constraint Programming in Oz

Oz is a concurrent constraint language with explicit concurrency, implicit synchronization, lexical scoping, mutable state, first-class procedures and first-class computation spaces. Smolka presents the computational model of Oz in [137]. Schulte, Smolka and Würtz describe in [132] the integration of constraint programming including encapsulated search in Oz.

This section gives an introduction to constraint programming in Oz. It explains the facilities to program propagation algorithms with finite domains (Section 3.2) and how to connect those propagation algorithms with computation spaces as means to realize speculative computation as search and constraint combinators (Section 3.3).

For a comprehensive introduction to Oz see [60] and for constraint programming in particular consult [131].

3.1 The Core

Values and Logic Variables The presented Oz programs compute on integers, literals and rational trees. A literal is either a symbolic value (called an *atom*) or a value without any structure but with an identity (called a *name*). A value is referred to by a *logic variable*. This is a variable which initially denotes no value until it is eventually bound. Values and logic variables are stored as a value graph in the constraint store (see Section 5.1.1 for details).

Constructor Values The atom `nil` denotes by convention an empty list. Other atoms, as `'|'` and `'#'`, are used by syntactic sugar such as mixfix tuples ($x_1\#\ldots\#x_n \equiv$ `'#'`$(x_1 \ldots x_n)$), cons-cells ($x_1|x_2 \equiv$ `'|'`$(x_1\ \ x_2)$), and lists (`[`$x_1 \ldots x_n$`]` $\equiv$ `'|'`$(x_1$ `'|'`$(\ldots$`'|'`$(x_n$ `nil`$)\ldots)))$. The length `N` of a list `L` can be computed by `{Length L N}` and dually, a list `L` of length `N` can be created by `{MakeList N L}`. Two lists `L1` and `L2` can be concatenated to `L3` by `{Append L1 L2 L3}`. Tuples and lists are called *vectors*. The notation $\langle x_1 \ldots x_n \rangle$ is used for a vector of n elements x_i.

Statements An Oz program consists of *statements* which are sequentially executed. The core statements are shown in Figure 3.1. A statement σ is empty (**`skip`**), declares a

σ	::= **skip**	*empty statement*
	| **local** x **in** σ **end**	*variable declaration*
	| x = y | x = v | x = e	*tell statement*
	| σ_1 σ_2	*sequential composition*
	| **thread** σ **end**	*thread creation*
	| **if** x **then** σ_1 **else** σ_2 **end**	*conditional*
	| **case** x **of** v **then** σ_1 **else** σ_2 **end**	*pattern matching*
	| **proc** {x $\overline{y}$} σ **end**	*procedure definition*
	| {x $\overline{y}$}	*procedure application*
e	::= x + y | x - y	*arithmetic expressions*
	| x == y	*equality test*
v	::= s	*scalar value*
	| $l(x_1 \ldots x_n)$	*tuple construction*
l	::= *atom*	*atom*
	| **true** | **false**	*Boolean names*
s	::= l	*literal*
	| *integer*	*integer*
x, y, z	::= *variable*	*variables*
$\overline{x}$	::= ϵ | x $\overline{x}$	*variable list*

Figure 3.1: Statements of Oz .

variables (**local** x **in** σ **end**[1]), tells constraints to the store (x = y *resp.* x = v), or composes two statements (σ_1 σ_2).

Oz provides syntactic sugar for local variable declaration in the σs of statements for thread creation, conditional and pattern matching. A local variable declaration **local** x **in** σ **end** can there be simplified to x **in** σ.

Expressions Expressions include the equality test and arithmetic expressions and have the obvious meaning.

Concurrency and Synchronization The creation of a thread is not synchronized and spawns a concurrent thread of computation. The statement **thread** σ **end** creates a thread with an empty stack and pushes a tasks for σ onto the stack. A running thread pops tasks from the stack and evaluates them. A thread vanishes as soon as the stack is empty. Suspending (or synchronizing) statements block the whole thread. An example for a synchronizing statement is the conditional statement **if** x **then** σ_1 **else** σ_2 **end**. It pushes σ_1 (σ_2) on the stack if x is bound to **true** (**false**). The conditional statement suspends if x is unbound.[2] In this way, it synchronizes on x.

Pattern Matching Pattern matching is very useful for programming with tuples and

[1] Oz provides anonymous variables denoted by "_" which need not to be declared and every occurrence is distinct to any other occurrence.

[2] Note that Oz is a dynamically typed language so that also a run-time type-error can be raised.

lists. The statement **case** x **of** v **then** σ_1 **else** σ_2 **end** synchronizes on x to be determined. The variables occurring in the pattern v are created and v is unified with x. If unification succeeds, the variables in v are bound and σ_1 is pushed on the stack of the current thread. In case unification fails, σ_2 is pushed on the stack.

First-class Procedures and their Application Procedures are first-class citizens in Oz, *i.e.*, they can occur in programs everywhere where a value can occur. The creation of a procedure is not synchronized. A procedure is created by **proc** {x $\overline{y}$} σ **end**. It binds x to a fresh name ξ and adds an entry $\xi : \overline{y}/\sigma$ to the *procedure store*. A procedure in the procedure store $\xi : \overline{y}/\sigma$ can be applied by {x $\overline{y}$} if x is bound to ξ. Procedure application synchronizes on x.

Oz provides syntactic sugar for function-style procedure (for short function) definitions

z = **fun** {x $\overline{y}$} **end** $\equiv$ **proc** {x $\overline{y}$ z} σ z **end**

Note that z does not occur free in σ. A function can be applied according to the extended expression syntax:

e ::= z ={x $\overline{y}$} *function application*

Expressions can be nested on parameter positions in procedures and functions. Instead of `X=[1 2 3] {P X}` one can use `{P [1 2 3]}`.

List Iterators The following list iterators occur in program examples. Note that `F` is a binary and `P` a unary procedure.

`Z={Map [`X_1 `...`X_n`] F}` $\equiv$ `Z=[{F `X_1`} {F `X_2`} ...{F `X_n`}]`
`{ForAll [`X_1`...`X_n`] P}` $\equiv$ `{P `X_1`} {P `X_2`}...{P `X_n`}`
`{ForAllTail [`X_1`...`X_n`] P}` $\equiv$ `{P [`X_1 `...`X_n`]} {P [`X_2`...`X_n`]}...{P [`X_n`]}`

For concise applications of the above-mentioned list iterators, procedures are created "on-the-fly". In Oz, the `$` transforms a statement to an expression which can be used to mimic anonymous procedures. For example `{Map [1 2 3] fun {$ X} 2*X end Y}` uses `$`[3] to pass the function to `Map` which eventually binds `Y` to `[2 4 6]`.

3.2 Finite Domain Constraints

This sections introduces finite domain constraints and explains how to implement propagation algorithms with them.

Basic Finite Domain Constraints A finite domain constraint $x \in d \subseteq \{0, \ldots, \sup_{\text{fd}}\}$[4] denotes an integer $i \in d$. Domain reduction removes elements from d until d is a singleton set $\{i\}$, *i.e.*, x refers to the integer i.

Basic finite domain constraints are imposed by the `::`-operator (Figure 3.2) where *setdescr* is an Oz value describing a finite set of integers.

[3] The function definition is turned into an expression whose value is the defined function.

[4] In Mozart, $\sup_{\text{fd}}$ is $2^{28} - 1$. The notation for a set $\{m, \ldots, n\}$ denotes a set containing all integers $i : m \leq i \leq n$.

σ	::= x `::` *setdescr*	*tell finite domain*

Figure 3.2: Imposing basic finite domain constraints.

setdescr	::= *set*	*set of integers*
	| `compl(`*set*`)`	$\{0, \ldots, \sup_{fd}\} \setminus set$
set	::= `nil`	$\emptyset$ *(empty set)*
	| *primset*	*primset*
	| `[`$primset_1 \ldots primset_n$`]`	$primset_1 \cup \ldots \cup primset_n$
primset	::= *integer*	$\{integer\}$
	| $integer_1$`#`$integer_2$	$\{integer_1, \ldots, integer_2\}$

Figure 3.3: Description of a set of integers.

The `::`-operator synchronizes on *setdescr* (Figure 3.3; the comments on the right show how a set is constructed) to be determined.

Propagators Mozart provides a rich set of predefined propagators for finite domain constraints including linear and non-linear (in-, dis-) equations, Boolean connectors, symbolic relations and scheduling constraints (see [46, Sections 5 and 6]) by the module `FD`. Individual abstractions provided by a *module* can be accessed by the `'.'`-operator, *e.g.*, the propagator for the *alldiff*-constraint `distinct` is obtained by `FD.distinct`.

finite domain operators	`=:`	`>:`	`>=:`	`<:`	`=<:`	`\=:`
corresponding relation	$=$	$>$	$\geq$	$<$	$\leq$	$\neq$

Table 3.1: Correspondence between finite domain operators and relations.

Equations are provided both by expressions and abstractions of the module `FD`. The operators for equations are shown in (Table 3.1). For example, the in-equality $2 \times \mathtt{X} - 3 \times \mathtt{Y} + 7 \times \mathtt{Z} \leq 5$ can either be represented by the finite domain expression `2*X-3*Y+7*Z=<:5` or by the abstraction `{FD.sumC [2 ~3 7] [X Y Z] '=<:' 5}`. Such abstractions are useful if equations are generated at run-time.

Mozart provides propagators with sophisticated filters, as for example an arc-consistent *alldiff*-constraint `FD.distinctD`[5] [120] or propagators for scheduling problems [6, 27, 28, 29, 150]. Such constraints are sometimes denoted by *global constraints* due to Beldiceanu who coined the notion in [11].

Reification Mozart provides propagators for *reified* constraints [143]. A reified constraint $(c \leftrightarrow b)$ of a constraint c reflects the validity of c to a 0/1-variable b: $(c \leftrightarrow b) \wedge b \in \{0, 1\}$. That means if c is entailed (dis-entailed) by the constraint store, b is constrained to 1 (0). Otherwise, if $b = 1$ (0), c ($\neg c$) is imposed.

[5]The propagator `FD.distinct` takes only advantage of determined parameters.

Propagators for reified arithmetic equation constraints are syntactically supported: a finite domain equation can simply be equated to a $0/1$-variable. For example, the reified version of `X+Y=:Z` is `X+Y=:Z = B` where `B` is a $0/1$-variable.

An Example Program Finite domain constraint programming implements the propagation, branching and exploration algorithms for a given problem in a *script*. Here, the propagation algorithm for the "send + more = money"-puzzle is programmed while the branching and exploration algorithms are postponed until Section 3.3.1.

$$\begin{array}{cccccc} & & s & e & n & d \\ + & & m & o & r & e \\ \hline = & m & o & n & e & y \end{array}$$

Every letter in the following equation denotes a decimal digit and s and m must be unequal to 0. The problem is to find an assignment for every letter that the equation holds. The puzzle is expressed in terms of constraints by:

$$e, n, d, o, r, y \in \{0, \ldots, 9\} \wedge s, m \in \{1, \ldots, 9\} \wedge \mathit{alldiff}(s, e, n, d, m, o, r, y) \quad (3.1)$$

$$\begin{aligned} & 1000 \times s + 100 \times e + 10 \times n + d \\ + & 1000 \times m + 100 \times o + 10 \times r + e \\ = \; & 10000 \times m + 1000 \times o + 100 \times n + 10 \times e + y \end{aligned} \quad (3.2)$$

Procedure `Money` implements these constraints (Program 3.1) and is part of the script for solving the problem. The procedure has the formal argument `Sol` which is constrained to the list of problem variables (`S`, `E`, `N`, `D`, `M`, `O`, `R`, and `Y`). The `ForAll`-loop constrains all problem variables to the domain $\{0, \ldots, 9\}$. The enclosed dis-equalities remove 0 from the domains of `S` and `M` and propagator `FD.distinct` enforces all its parameters to be pairwise distinct (constraint (3.1)). Finally, the equation (`=:`) imposes the central constraint of this puzzle (constraint (3.2)).

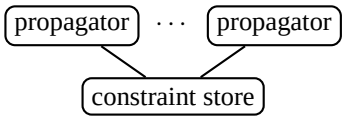
```
proc {Money Sol} [S E N D M O R Y] = Sol in
   {ForAll Sol proc {$ V} V :: 0#9 end}
   S \=: 0  M \=: 0  {FD.distinct Sol}
                1000*S + 100*E + 10*N + D
              + 1000*M + 100*O + 10*R + E
   =: 10000*M + 1000*O + 100*N + 10*E + Y
end
```

Program 3.1: The implementation of the propagation algorithm for "send+more=money"-problem.

3.3 Computation Spaces

Computation spaces are a concept to encapsulate speculative computation. Oz provides spaces a first-class citizens and thus, makes it possible to program tree search (Section 3.3.1) and constraint combinators (Section 3.3.2) in Oz itself. Schulte gives in his thesis [128] a thorough account on design, implementation and application of computation spaces.

Overview A *computation space* hosts a constraint store and propagators connected with this store and thus, constraint propagation as described in Section 2.3 takes place in a computation space.



A computation space has a *root variable* and a *status*. The root variable is used to access the constraint in the space. The status indicates whether constraint propagation is currently running or not, and if not running, (*i.e.* the space is stable) whether the space is succeeded, failed or distributable. A distributable space contains a *distributor* which represents a branching constraint (explained in Section 3.3.1). Additionally, the state can also indicate that a space has been merged with another space. A space is failed if one of its propagators failed. A space is succeeded if it is neither failed nor distributable. A succeeded space is stable. The intuition is that a stable space cannot be made unstable from computation outside of the space. It is essential for programming to detect *stability* of a space to obtain predictable execution behavior.

Combining spaces results in a tree-shaped hierarchy of computation spaces, short *space hierarchy*.[6] The root of a space hierarchy is called the *top-level space* or *root space*. In contrast to other computation spaces, a top-level space becomes never failed. As a consequence of this, domains of variables created in the top-level space, so-called *top-level variables*, can never become empty.

Operations Operations on spaces are provided by the module `Space`. The following operations on spaces are regarded.

`Space.new`: *Script* $\rightarrow$ *Space*
: A space is created by `{Space.new` x y`}` which synchronizes on x to be bound to a value which is supposed to be a unary procedure (a script). Then a fresh space s is created and y is bound to a fresh name ξ which is used to refer to s ($x : \xi \mapsto s$). The root variable of the space is initialized with a fresh variable z. Finally, a thread in s is created to run `{`x z`}` in the new space.

`Space.ask`: `{Space.ask` x y`}` *Space* $\rightarrow$ *Atom*
: The state of a space s can be inspected by `{Space.ask` x y`}` which synchronizes on $x : \xi \mapsto s$ and on s being stable or merged. Then y is bound to either the atom `failed`, `merged`, `alternatives(`n`)`, or `succeeded` depending on whether the space is failed, merged, contains a distributor with n choices or neither failed, merged nor distributable, respectively.

`Space.clone`: *Space* $\rightarrow$ *Space*
: A space is cloned by `{Space.clone` x y`}` which synchronized on $x : \xi \mapsto s$ and on s being stable. Then y is bound to a clone $\xi' \mapsto s'$ where ξ' is a fresh name and all names and variables in s' are consistently renamed to fresh names and variables.

`Space.commit`: *Space* $\times$ *Integer*
: A distributable space is committed to an alternative by `{Space.commit` x y`}` which synchronized on $x : \xi \mapsto s$, on s being stable and not merged. Additionally,

[6] The notion "hierarchy" is used to distinguish from search tree created with computation spaces.

it synchronized on y to be bound to an integer. An error is raised if s is not distributable or y does not denote a valid alternative (explained in Section 3.3.1).

`Space.merge`: *Space* $\rightarrow$ *Any*

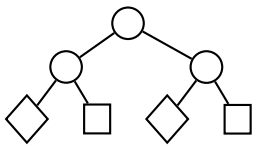
A space is merged with the current space by `{Space.merge` x y`}` which synchronizes on $x : \xi \mapsto s$. Merging equates the root variable z of s with y, *i.e.*, $z = y$. The application of merging in this thesis is restricted to only access the root variable of a space.

3.3.1 Search

This section presents copy-based search engines based on computation spaces. A simple search engine is implemented and applied to the example of Section 3.2. This section closes by presenting some predefined search engines of Mozart.

Copy-based Search Search is applied to a problem if it cannot be solved only by constraint propagation. Performing a case analysis on the problem adds branching constraints to the problem. Thus, the constraints of the problem are strengthened and new propagation is triggered. But this propagation may fail. Hence, *state restoration* has to be used to recover from failure. Spaces suggest a copy-based state restoration approach in contrast to widely used trailing approaches [2, 39]. Search engines based on spaces were first presented by Schulte and Smolka in [130] and by Schulte in [127].

A branching constraint can be expressed by Oz's choice combinator **`choice`** σ_1 `[] ... []` σ_n **`end`** where σ_i denotes an alternative constraint. As an example, regard **`choice`** `X=1 [] X \=: 1` **`end`** which imposes on one branch of the search tree $x = 1$ and on the other one $x \neq 1$. A choice combinator acts as distributor in a space and makes a space distributable. The figure on the right denotes a a search tree. A circle node denotes a choice point node (distributable space), a square node a failed node and a rhombus node a solution.



A tree of computation spaces created by copy-based search is in contrast to hierarchies of spaces where branches are dependent on each other while spaces in a search tree are independent from each other (Section 3.3.2).

A Search Engine A copy-based search is illustrated by implementing a search engine which searches for all solutions. This engines is very simple search engine. It is restricted to binary search trees and cannot be interrupted.

The search engine is shown in Program 3.2. Function `SearchEngine` expects a space as argument and returns a list of solutions, *i.e.*, root variables of solution nodes. First, the state of the argument space is obtained using `Space.ask`. Since `Space.ask` synchronizes on `S` to be stable, search starts when no computation has an impact on the state of `S`. If the space is distributable, is it cloned and one alternative is committed in one space and the other alternative in the other space. This triggers constraint propagation. Then, the search engines is recursively applied to the committed spaces.

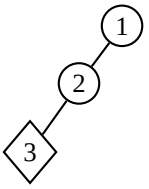
If space `S` is not distributable, it is checked whether it is succeeded or not. If so, the root variable denotes a solution which s retrieved by merging the solution space with the

space the engines runs in. If S is not succeeded, an empty solution is returned indicated by `nil`.

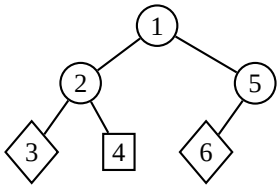
```
fun {SearchEngine S} A = {Space.ask S} R in
   if A==alternatives(2)              % space S is distributable
   then S1 = {Space.clone S} in                % clone space S
      {Space.commit S 1}                  % commit alternatives
      {Space.commit S1 2}
      {Append {SearchEngine S1} {SearchEngine S} R}
   else
      if A==succeeded                             % solution found
      then Sol={Space.merge S} in R=[Sol]    % retrieve solution
      else R=nil end                                % no solution
   end
   R
end
```

Program 3.2: A search engine for finding all solutions.

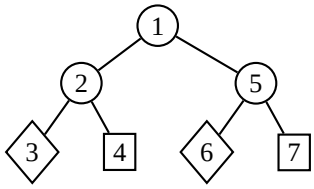
Branch-and-bound Search It is often the case that a best solution according to some criterion is asked for. This criterion is expressed by an ordering constraint. The presented search engine searches for all solutions no matter if they are better than previously found ones or not. *Branch-and-bound search* cures the problem. As soon as a solution is found, branch-and-bound imposes on all open nodes of the search tree the constraint that newly found solutions have to be better than the last one according to some criterion. An open node is a choice point node with less than 2 edges.



(a) Finding the 1st solution.



(b) Finding the 2nd solution.



(c) The tree is completely explored.

Figure 3.4: An evolving search tree..

Figure 3.4 shows how a search tree is created. In Figure 3.4(a), the first solution is found in node 3. An ordering constraint is added to the open nodes 1 and 2 which enforces that the next solution is a better one. The next solution is found in node 6 (Figure 3.4(b)) and an improved ordering constraint is added to the open node 5. Eventually node 7 fails, the search tree is completely explored and the solution found in node 6 is the optimal solution (Figure 3.4(c)). Branch-and-bound guarantees that last solution is the best and the ordering constraint prunes the search tree additionally.

Example Continued The example of Section 3.2 is continued by adding the branching and exploration algorithms (Program 3.3).

```
proc {MoneyScript S}
   {Money S}
   {FD.distribute naive S}
end
```

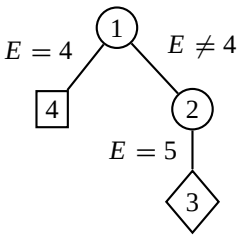
Program 3.3: Complete script for the "send+more=money"-problem.

The finite domain library of Mozart provides the abstraction `FD.distribute` which combines branching and exploration algorithms for finite domain constraints. This abstraction can be configured for various branching and exploration strategies by its parameters. Further, a procedure can be passed to `FD.distribute` which is run on stability. This is useful for sophisticated propagation algorithms (see for example Section 14.3.1). In this case, a naïve strategy is chosen, which takes the first undetermined variable $x \in d_n$ in `S` and creates a distributor **`choice`** $x = n$ `[]` $x \neq n$ **`end`** where n is the minimal value of d_n.

The script `MoneyScript` is submitted to the previously defined search engine `SearchEngine`:

```
Sol = {SearchEngine {Space.new MoneyScript}}
```

and `Sol` is bound to the solution `[[9 5 6 7 1 0 8 2]]`. The search tree of the problem is shown in Figure 3.5.



node	S	E	N	D	M	O	R	Y
1	9	{4, . . . , 7}	{5, . . . , 8}	{2, . . . , 8}	1	0	{2, . . . , 8}	{2, . . . , 8}
2	9	{5, . . . , 7}	{6, . . . , 8}	{2, . . . , 8}	1	0	{2, . . . , 8}	{2, . . . , 8}
3	9	5	6	7	1	0	8	2

Figure 3.5: The search tree and the constraints in the nodes of the "send+more=money"-problem.

The edges of the search tree are annotated with the branching constraints imposed by the distributor. The table shows the constraints at the problem variables in the respective nodes. As one can see, `S`, `M`, and `O` are determined just by constraint propagation.

Predefined Search Engines Mozart provides two kinds of predefined search engines: an interactive one, the Oz Explorer [126] (Figure 3.6), and various non-interactive ones, *e.g.*, `SearchOne` to find the first solution, `SearchBest` to find the best solution by branch-and-boundsearch, and `SearchAll` to find all solutions. These machines have in common that the problem formulation is passed as a unary procedure called *script*.

These search engines provide control and debugging facilities. For example, search can be interrupted or the size of the search tree can be restricted. The Explorer makes it possible to explore a search tree in an incremental way and to inspect the individual nodes (see [129] for more control options). Figure 3.6 shows the Oz Explorer window for the "send+more=money"-problem and the tree is identical to the tree in Figure 3.5.

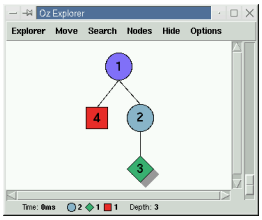


Figure 3.6: Oz Explorer.

3.3.2 Constraint Combinators

Constraint combinators are another way of to facilitate speculative computation in Oz. They provide extra expressiveness for constraint programming. In contrast to search, combinators create hierarchies of computation spaces. That means, computation spaces depend on each other. The constraints of a superordinated space are promoted to the subordinated space while constraints in the subordinated space are encapsulated (Figure 3.7). The arrows in the figure denote the direction of constraint visibility. In search, the spaces are independent from each other, *i.e.*, there is no promotion of constraints between nodes in a search tree since branches are cloned spaces.

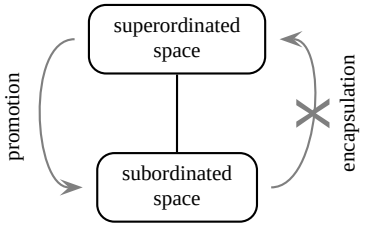


Figure 3.7: Constraint promotion and encapsulation.

The implementation of constraint combinators requires in general a more precise level of control than offered by stability. Schulte discusses these issues in detail in [128, Chapter 11]. But for the example in this section, stability is an sufficient means of control.

Example: A Negation Combinator The implementation of a constraint combinator is illustrated by the implementation of a negation combinator. Stability is sufficient for this combinator since a failed *resp.* succeeded space is stable. The negation combinator is shown in Program 3.4.

```
proc {Negation P} N = {Space.new P} in
   failed = {Space.ask N}
end
```

Program 3.4: A negation combinator.

First, the argument passed to procedure `P` is executed in a newly created space `N` and then, the state of `N` is constrained to `failed`. As soon as `N` is stable, `Space.ask` synchronizes and the combinator fires, *i.e.*, succeeds if `N` is failed and fails if `N` is succeeded.

Figure 3.8 demonstrates how the negation combinator works. The combinator is run in a space S with the constraints $x \in \{1, \ldots, 10\}$ and $y \in \{1, \ldots, 10\}$ and creates the space N with the propagator $x < y$. The propagator constrains the domains of x and y in N which is invisible in S due to encapsulation. Then, $x > 5$ is imposed in S which prunes the domain of x and y. This is promoted to N and the domains of x and y are

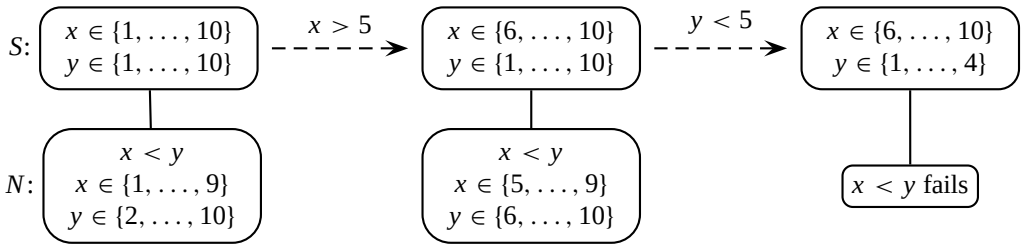


Figure 3.8: Using the negation combinator.

further pruned due to $x < y$. But again, encapsulation prevents this changes to be visible in S. Finally, $y < 5$ is imposed and the changes to the domains of x and y are promoted to N which fails $x < y$. Now space N becomes stable and `Space.ask` synchronizes. The combinator fires and succeeds.

Part I

Constraint Propagation Engines

Chapter 4

A Propagation Engine Architecture

This chapter presents an architecture of a sequential propagation engine which realizes the computational model discusses in Section 2.3.

4.1 Components of a Propagation Engine

A propagation engine consists of *domain-independent* propagation services and *domain-dependent* domain solvers. A single propagation engine can be connected to several domain solvers which may cooperate with each other (for example, see Section 13.1.3 for connecting finite domain and finite integer set constraints).

The separation of a propagation engine in propagation services and domain solvers suggests to introduce of an interface which is done in Chapter 6.

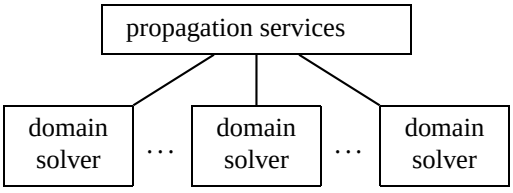


Figure 4.1: Propagation engine = propagation services + domain solvers.

A propagation engine is hosted in a computation space. The architecture reflects this by providing means to compute the status of a computation space.

4.2 Representation of the Constraint Graph

As discussed in Section 2.3, running the propagation engine results in transforming the constraint graph for a propagation algorithm defined by a constraint program (Section 2.1). This section discusses the representation of the constraint graph while the remaining sections of this chapter discuss the transformation of the constraint graph.

A constraint graph is represented in this model by an *engine constraint graph* consisting of variable nodes and propagator nodes.

4.2.1 Constraint Variables

A constraint variable (Figure 4.2) represents a variable node of the constraint graph in Figure 2.3 on page 13. It consists of a *variable head* and a *variable body*. The head is domain-independent and is connected to the body. The body consists of the domain d and different sets of connected sleeping propagators, short *sleeping propagator sets*. These sets are indexed by events e_1 to e_n. An event e_i on the variable causes the propagators in the sleeping propagator set indexed by e_i to be scheduled.

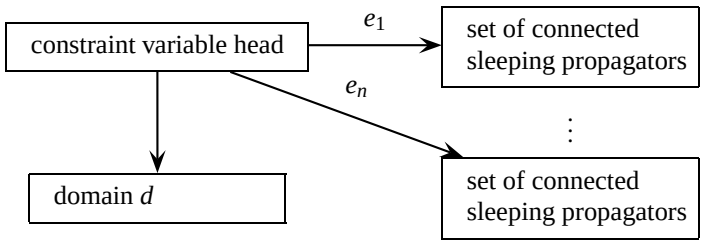


Figure 4.2: A Constraint Variable.

The domain representation and the indexing events have to be defined by the domain solver while the representation of the sets of sleeping propagators is part of the propagation services.

4.2.2 Propagators

A propagator represents a propagator node of the constraint graph. It is an entity with a shared and a private state (Figure 4.3). Additionally, it is equipped with a propagation function which computes new basic constraints for the parameters. This function is explained in detail in Section 4.3.

Shared and Private State The *shared state* comprises the propagator's parameters, which are typically shared between several propagators. The *private state* consists of the *propagator head* and the *propagator body* and its information is accessible only by the propagator itself. The body is defined by the domain solver. It holds the references to the parameters and provides the *propagation function*.

Role of the Propagator Head The propagator head is the domain-independent interface between the propagation services and the corresponding propagator body. The head has access to all information needed to manage and execute a propagator, as for example the execution state and (via the body) the propagation function (see Section 4.4).

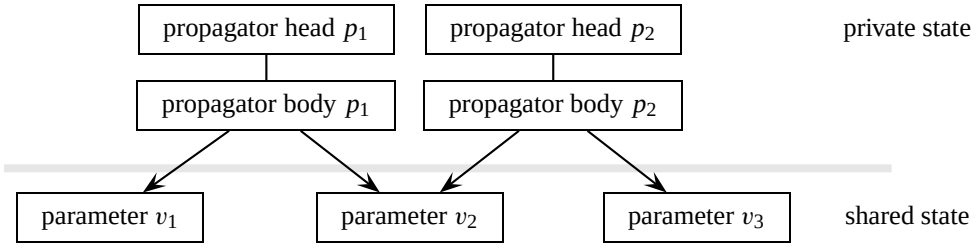


Figure 4.3: Propagators with private and shared state.

4.3 Propagator Execution

Executing a propagator p imposing a constraint c means to run its *propagation function*. This function computes by its filter F_c new domains d' strengthening the current domains d of the propagator's parameters. Propagation events are computed from d' and a *constraint profile* taken from d. A constraint profile $prof_d$ of a domain d contains characteristic information of d to compute events in conjunction with a strengthened domain d'. A profile $prof_d$ is supposed to use less memory then the whole domain d. Constraint profiles are domain-dependent and hence, part of the domain solver. The structure of a propagation function is shown in Figure 4.4.

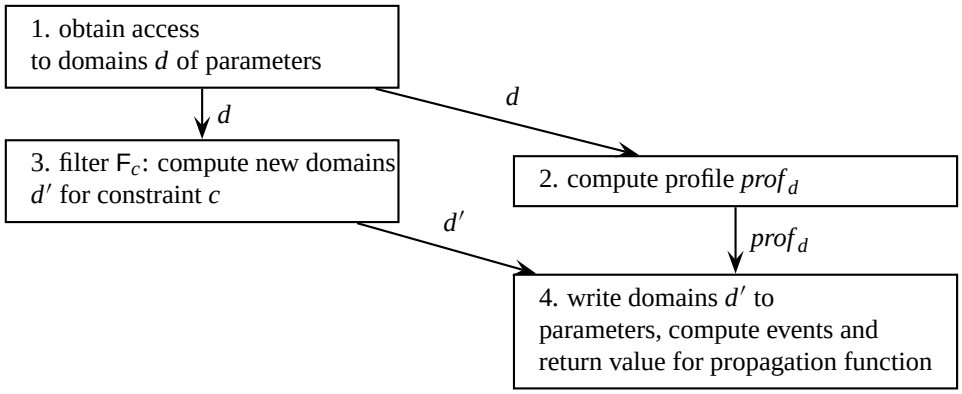


Figure 4.4: Propagation function of a propagator for constraint c.

A propagation function proceeds in four steps:

1. The propagation function obtains access to the domains and sleeping propagator sets of the parameters via the references stored in the body of the propagator.
2. Constraint profiles of d are taken.
3. New domains d' are computed by the filter F_c. A filter can additionally create new propagators on its parameters, for example to replace itself.
4. The new domains d' are written to the parameters. Events caused by filtering are computed from the constraint profiles $prof_d$ and d'. The computed events are passed together with the return value of the propagation function to the propagation

services. The return value indicates whether constraint c realized by p_c is entailed by the store (`entail`), inconsistent with store (`fail`), or none of both (`sleep`).

Constraint profiles are introduced to allow filters to override domains by providing memory-efficient means to memorize a previous state of a domain.

Running filters as part of executing propagator results in transforming the constraint graph by removing (propagator entailment), adding (propagator creation), and invalidating the nodes (propagator failure).

4.4 Managing Propagators

Performing propagation on a constraint graph transforms the graph as long as events arise. Therefore, the propagation services maintain three sets of propagators as shown in Figure 4.5

Maintaining Sets of Propagators The *sleeping propagator set* is the union of the sleeping connected propagator sets of all variables. As soon as a propagator is scheduled by some event it is moved to the *runnable propagator set* (transition `schedule`). A propagator is returned to the sleeping propagator set if executing its propagation function returned `sleep` (transition `sleep`). Otherwise, in case the propagation function returns either `fail` or `entail`, the propagator is moved to the discarded propagator set (transition `discard`).

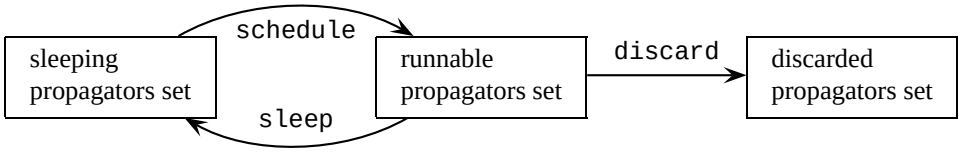


Figure 4.5: Sets of propagators maintained by the propagation services.

This set model makes it easily possible to detect if the computation space hosting a propagation engine is *`stable`* or *`entailed`* when propagation reaches a fixed-point. A fixed-point is reached if the runnable propagator set is empty. A computation space is *`entailed`* if the sleeping propagator set is additionally empty. Otherwise, the computation space is *`stable`*.

Managing an Individual Propagator The propagator management from the view point of an individual propagator p by the propagation services is as shown in Figure 4.6 and resembles the execution states in Figure 2.4 on page 13.

As soon as a propagator is created its propagation function is executed to perform an initial consistency check. According to the return value the propagator is moved to either the sleeping propagator set or discarded propagators set. In case the return value is `fail`, the propagation terminates immediately and the status of the host space is set *`failed`*. The transition from the sleeping to the runnable propagator set is caused by some event on a parameter of propagator p.

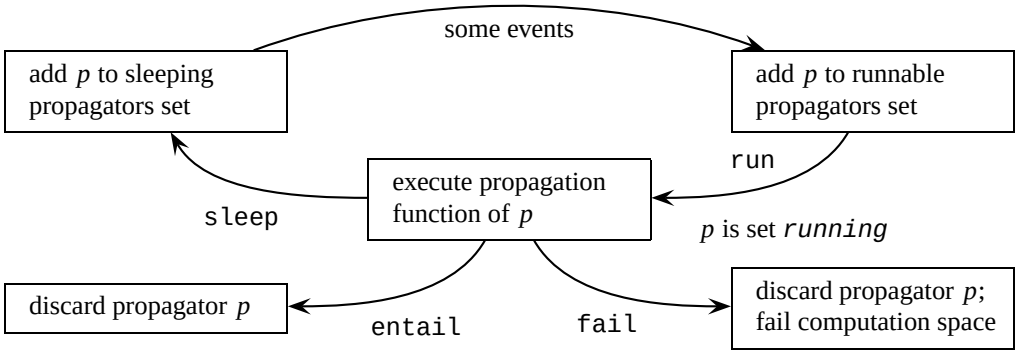


Figure 4.6: Managing an individual propagator.

The propagation services use a *propagator scheduler* to pick a propagator from the runnable propagators set and to set it *running* (transition `run` in Figure 4.6). The propagator scheduler guarantees that every propagator in the runnable propagator set is eventually run but does not make any assumption about when and in which order.

Non-monotonic Propagators Non-monotonic propagators occur in real-world applications, as for example in scheduling where tasks have to be allocated on resources. The pruning of a non-monotonic propagator depends on the current state of the constraint store. For example, Würtz presents in [150, Section 5.2.3] a non-monotonic filter algorithm which allocates a given task relative to a set of tasks (a so-called *task interval* [27]) depending on the current constraint store. Hence, to ensure that always the same propagation fixed-point is reached, non-monotonic propagators have to be invoked on the same basic constraints in the store.

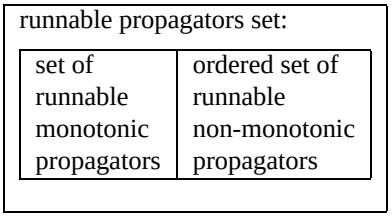


Figure 4.7: Extended runnable propagators set.

This is achieved by executing non-monotonic propagators in a fixed order after all monotonic propagators have reached a fixed-point. Therefore, every non-monotonic propagator is assigned a unique *priority* on its creation. The priority is stored in the head of the propagator and determines the order of execution. Further, the runnable propagators set is extended by an *ordered set of runnable non-monotonic propagators* (Figure 4.7). Monotonic propagators are stored the *set of runnable monotonic propagators*. The propagator scheduler runs all propagators in the set of runnable monotonic propagators before it runs all propagators in the ordered set of runnable non-monotonic propagators according to their priority. This guarantees that starting from the same constraint store, every execution of a non-monotonic propagator function faces the same basic constraints.

Note that running non-monotonic propagators may schedule other (monotonic and non-monotonic) propagators. Hence, propagation continues until both propagator sets are empty.

Chapter 5

Propagation Services for Mozart

This chapter discusses the integration of propagation services into the virtual machine (VM) of Mozart according to the architecture introduced in Chapter 4. The integration of propagation services is presented in two steps: (i) without spaces and (ii) with spaces. Step (i) starts by introducing Mozart's VM with threads and synchronization but without hierarchies of computation spaces in Section 5.1 and continues by discussing the integration of propagation services in this VM in Section 5.2. Step (ii) takes hierarchies of computation spaces into account is presented in Section 5.3. The VM is extended by computation spaces in Section 5.3 and the following sections adapt propagation services for this extended VM. The chapter closes with a discussion in Section 5.4.

5.1 Mozart's Virtual Machine

This section describes the part of Mozart's VM needed for concurrent computation with synchronization. The VM provides a computational environment to the propagation services including the constraint store (for hosting constraint variables), concurrency and synchronization (for scheduling and running propagators), and foreign code execution (for propagator creation).

The Mozart VM is explained in great detail in [88, 124, 89] while the integration of spaces is presented in [128].

5.1.1 Constraint Store

The constraint store stores values as a *value graph*: A node represents the current knowledge about a value while an edge denotes the equality between values. A node can either be a *value node* representing a value, a *variable node* representing a variable (a value still unknown), or a *reference node* referring to another node. Every node carries its type. A variable node represents the head of a variable and points to the corresponding body. A variable is bound to a value by overriding the corresponding variable node by a reference node pointing to the corresponding value node.

5.1.2 Concurrency and Synchronization

The Mozart VM provides a sequential implementation of concurrency. Further, concurrent computation can synchronize on the binding of variables.

Concurrent Computation A *thread* executes a sequence of statements $\sigma_1, \ldots, \sigma_n$. These statements are stored on a stack. A thread runs by taking the top-most statement from the stack and executing it.

The execution of threads is controlled by the *scheduler* of the VM which runs at most one thread at a time. Thereby, threads go through different states (similar to propagators). The state of a newly created thread is `runnable`. A runnable thread will be eventually executed (due to fairness). Assigning a time-slice to a runnable thread sets the state to `running` and runs the thread. A running thread is preempted if its time-slice is expired and the thread is not terminated, *i.e.*, its stack is not empty. Preemption is needed for fairness to eventually run all runnable threads. A preempted thread is immediately set `runnable` again.

Synchronized Computation The VM implements synchronization as follows: If a thread τ executes a statement σ suspending on a variable v, then the thread is *suspended* and its state is set `blocked`. Then, a reference to τ (called *suspension*) is added to the *suspension set* of v and σ is pushed back on τ's stack. The suspension set is stored in the body of variable. Since the top-most statement of τ's stack is σ, resuming τ resumes σ. A blocked thread is set `runnable` again by *scheduling* it.

Binding variable v causes all threads in the suspension set of v (among them thread τ) to be scheduled and the corresponding suspension to be *removed* from the suspension set of v. To avoid that binding different variables may cause a single thread to be scheduled more than once, only threads with state `blocked` can be scheduled.

Execution of Foreign Code The VM is able to run foreign code by *foreign functions*. The VM provides a uniform calling scheme for passing arguments to and obtaining the execution status from a foreign function. Further, a foreign function can suspend on a variable v. In that case, v is passed to the VM and the statement calling the foreign function is suspended as described above. Note that a foreign function *cannot* be preempted.

5.2 Propagation Services

Propagation services are based on the architecture presented in Chapter 4 and extend the Mozart VM by the domain solver-independent part of propagation engines (see Figure 4.1 on page 29). This makes it possible to create a propagation engine by attaching the corresponding domain solver to the VM.

Propagation services extend the VM by constraint variables (Section 5.2.1), propagators (Section 5.2.2) and propagator management (Section 5.2.3).

5.2.1 Constraint Variables

A *constraint variable* is derived from a variable according to Section 4.2.1:

$$\text{constraint variable} = \text{variable} + \text{domain } d + \text{event sets}.$$

In the following, the operations required to integrate constraint variables in the Mozart VM are discussed.

Telling Basic Constraints In contrast to binding a variable to a value, as discussed in Section 5.1.1, a constraint variable can be successively constrained until it denotes a single value. This is simply done by updating the domain d of a constraint variable with a strengthened domain d'. Connected propagators waiting for caused events have to be scheduled.

Computing Events and Scheduling Propagators Constraint propagation on constraint variables causes events and as consequence, connected propagators waiting for the respective events have to be scheduled. As discussed in Section 4.3, efficient computation of events uses constraint profiles.

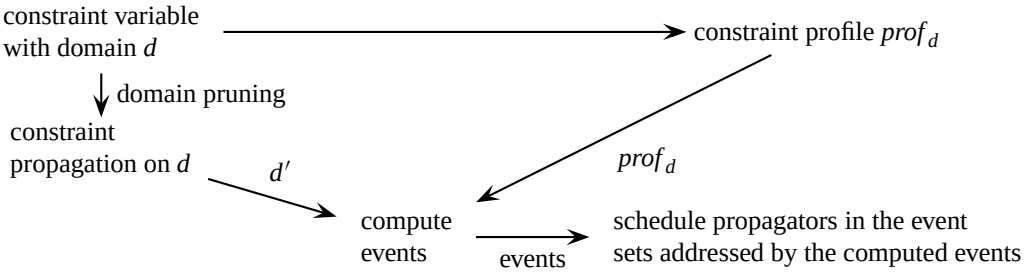


Figure 5.1: Relation between basic constraints, constraint profiles, and events.

Figure 5.1 shows how a domain, a constraint profile, and events interact. Before constraint propagation, a constraint profile $prof_d$ is taken from d. After constraint propagation has finished, the strengthened domain d' is compared with $prof_d$ to compute the events for scheduling sleeping computation. Eventually, the propagators in the respective sleeping propagator sets are scheduled.

Note that actually every domain modification during domain pruning produces events. To avoid frequent event computation, events are computed once when pruning has finished. Constraint profiles make it possible to override domains during pruning.

A pesimistic estimation for a profile is to memorize the complete domain before propagation. Profiles are mainly used where many domains are under consideration at once, as in propagation functions of propagators. An example for a finite domain constraint profile and the computation of the corresponding events is given in Section 8.2.

5.2.2 Propagators

A propagator consists of a head and a body (Figure 4.3 on page 31). The head is domain independent and stores all information for managing a propagator. The body of

a propagator is defined by the domain solver and is private to the propagator. It stores the references to the parameters which are shared with other propagators. Additionally, other data structures, as tables of pre-computed values, can be shared between different propagators. The private state can also be used for optimizations, *e.g.* storing intermediate results to avoid their re-computation.

Execution and Execution States of a Propagator A propagator performs constraint propagation by executing its *propagation function*. The propagation function is defined by the domain solver and its return value indicates the VM to what execution state to proceed from state `running` (Figure 2.4 on page 13). The current execution state of a propagator is stored in the private state of the propagator.

Creation of a Propagator A propagator is created in three steps as shown in Figure 5.2. The code for creating a new propagator is run by a foreign function. Such a function is called a *creator function*.

1. *Check expected constraints:* If the type of the parameters is incorrect raise a type-error. If the parameters have not the expected mode suspend the creator function on the responsible parameters. This cause the creator function to be re-invoked when additional information becomes available.
2. *Create propagator:* Create the propagator and initialize the propagator's state in particular with references to the parameters.
3. *Impose propagator:* The propagation function of the propagator is initially run. If this run does not terminate the propagator (by returning `fail` or `entail`), references to the propagator (called *propagator suspensions*) are added to sleeping propagators sets of the parameters. These sets correspond to the events the propagator is scheduled on the respective parameters.

Figure 5.2: Steps of creating a propagator.

A propagator can only be imposed on parameters of expected type and mode. Parameters must have the expected type since for example a finite domain propagator can obviously not cope with finite sets of integers. Even if no type-error occurred, the parameters must have the expected mode to make it *impossible* to invalidate constraint propagation. For example, if the parameters of a finite domain propagator are not yet finite domain variables, its propagation might be wasted when the parameters are later on constrained to some values incompatible with finite domains. Type and mode are determined by the constraints expected at the parameters, called *expected constraints*.

Non-monotonic Propagators On creation, a non-monotonic propagator is assigned a unique priority by the propagation services which is stored in the propagator's head. A propagator is monotonic if there is no priority stored. This is the default case.

Propagation Function Constraint propagation is performed by a propagation function as shown in the diagram in Figure 4.4 on page 31. This diagram is made linear by

collapsing steps 1 and 2 such that constraint propagation proceeds in three steps (Figure 5.3).

1. *Read parameters:* A propagator's body stores reference nodes to its parameters. The propagation function obtains access to the domain of a parameter by a *access variable*. During creation of an access variable, the domain of the corresponding parameter is obtained and the constraint profile is computed.
2. *Perform constraint propagation by filtering:* This part computes the actual elements to be removed from the domains of the parameters. Access to the domains is obtained by the corresponding access variables.
3. *Write parameters:* This part is responsible to write back the domains computed by the previous step. Additionally, the events caused by constraint propagation are computed (using the constraint profiles stored in the access variables) and propagators waiting for events are scheduled.

Figure 5.3: Steps of running a propagation function.

5.2.3 Propagator Management

This section describes the integration of propagator execution as part of propagation services into the Mozart VM. This design of the integration is based on an analysis of real-world constraint programs and thus, the requirements of realistic applications are met.

Analyzing a Propagator's Life-cycle Propagators are long-lived, *i.e.*, they are re-invoked many times. Running the constraint application programs of the Mozart test suite[1] produces the figures in Table 5.1. These figures are a rather pessimistic estimation since the application programs in the test suite try to provoke "pathological" situations where global variables are typically involved.

propagators	**immediately**		**remaining**
created	**failed**	**entailed**	**propagators**
232.793	3.461	106.421	122.911

Table 5.1: Results of initial runs of propagation functions.

The 122.911 remaining propagators are neither immediately failed nor entailed. They need 4.742.490 re-invocations to eventually become entailed or failed. That means a propagator which is not immediately failed or entailed, is re-invoked on average 38 times.

[1]The Mozart test suite is a collection of dedicated test cases and constraint programs for practical application problems. The Mozart test suite is part is the source code distribution of Mozart [99].

Initial Run of Propagation Function The figures in Table 5.1 shown that about 47 % of all propagator immediately terminate. This is the reason for the initial run of the propagation function while a propagator is created. Thus, by early detection of propagator termination adding propagator suspension to the parameters can be avoided (see step 3 in Figure 5.2).

Persistent Propagator Suspensions Due to the typically large number of re-invocations of propagators, frequent changes to the constraint graph (removal and addition of labeled edges; see Figure 2.3 on page 13) would be inefficient. Hence, the graph is kept by making propagator suspensions *persistent*, *i.e.*, a propagator suspension is not removed from an event set if the propagator is scheduled. Instead, it is left in the event set and since it refers to a scheduled propagator, multiple scheduling of the propagator is avoided.

Propagator Execution The VM is extended by a runnable propagator set (RP) and a non-monotonic runnable propagator set (NRP). These sets correspond to the set of runnable monotonic propagators *resp.* to the ordered set of non-monotonic propagators in Section 4.4. Scheduling a monotonic propagator adds the propagator to RP while a non-monotonic propagator is correspondingly added to NRP. As soon as RP is not empty, a thread T_{RP} is created which runs a *propagation engine*. The propagation engine executes the propagation functions of the propagators in RP. The thread T_{RP} can be preempted between two function invocations but is immediately scheduled again. As soon as RP is empty, T_{RP} terminates and the non-monotonic propagators in NRP are run by the *propagation engine for non-monotonic propagators* in an fixed order according to their priority. This propagation engine works identically to the monotonic propagation engine except that the order of propagators is maintained. Running non-monotonic propagators may in turn schedule monotonic propagators and thus, re-invoke the monotonic propagation engine.

5.3 Hierarchical Computation Spaces

This section integrates propagation services into the Mozart VM extended by hierarchies of computation spaces. Section 5.3.1 adds computation spaces to the VM presented in Section 5.1. Section 5.3.2 discusses the encapsulation of constraint propagation while Section 5.3.3 reconsiders propagators management for hierarchical computation spaces.

Hierarchies of computation spaces are of relevance for constraint combinators (Section 3.3.2). Copy-based search is completely orthogonal to space hierarchies since it transparently copies computation spaces (including constraint variables and propagators) without modifying the copied spaces (except consistent renamings of the entities in a copied space).

5.3.1 Computation Spaces in the Virtual Machine

A computation space (for short space) encapsulates computation. This means that computation inside a space S does not affect computation outside S. In contrast, computation outside S may have an effect on S. Especially, an inconsistency in a space fails only the space (including its subordinated spaces, if there are any).

Simulating Multiple Stores The VM maintains a single constraint store. Multiple stores of multiple spaces are simulated by a combination of *trailing* and *scripting*. Possible changes to a store are variable bindings to variables not hosted by the current space. These bindings are recorded in a *trail*. The VM has a single trail which consists of *trail entries*. A trail entry stores a variable node and the location of the variable node.

Whenever a trailed binding is undone, the binding is recorded in a *script* to be able to reconstruct the state of the computation space. Hence, every space has a script for its own. The script entry of a script memorizes the binding of a variable by storing the location of the variable node and the location of the node of the value the variable was bound to.[2]

Situatedness Encapsulated computation makes threads and variables *situated* by assigning them to a *home space*. Hence, every variable and thread stores its home space. The home space is typically the space of creation.[3] Changes to variables are only visible to the sub-hierarchy of the space hierarchy with the home space as root. A thread is executed in its home space. This causes computation to move between computation spaces. This requires to *install a space* when computation enters the space and to *de-install a space* when computation leaves the space. A variable v is called *local* to a space S if S is the home space of v. A variable is called *global* to a space S if S is not the home space of v and the home space of v is on the path from S to the root space.

Installation restores all bindings present before the space was left. The bindings are stored in the script and are reestablished by unifying the two components of the script entries. De-installation reverts all trailed bindings (called *untrailing*) by writing back the previous values to the memorized location according to the trail entries. At the same time the script is written to make the re-installation of these bindings possible whenever computation enters this space again.

Synchronized Computation Revisited Situatedness has to be taken into account by synchronization. When a variable is bound, only those threads are scheduled which are situated in the current space sub-hierarchy. A *current space sub-hierarchy* includes all spaces which are connected with root via the current space and the current space itself. Note that threads *must not be* scheduled when installing a space since this may cause infinite computation (see [128, Example 13.5]).[4]

[2] The location of the value node is stored because the value can be a variable which might be bound before the next reconstruction.

[3] Note *merging* two spaces makes the home space different to the space of creation (see Schulte's thesis [128, Section 13.4] for details).

[4] Tree constraints are an exception since they may add new bindings. See [128, Section 13.6.2] for details.

Status of a Space The VM needs to detect the status of a computation space which is either `failed`, `stable`, or `entailed`. A space is failed if a statement attempts to write a value to the store being in contradiction with the current values in the store. A space is stable if there are no runnable threads (detected by the scheduler of the VM) and no threads suspending on global variables ([128, Section 13.3]). Otherwise, outside computation would be able to trigger new computation within a space. A space is entailed if it is stable and no threads are left. This is detected by having a counter keeping track of the number of threads.

5.3.2 Encapsulation of Constraint Propagation

The introduction of space to the VM makes constraint variables situated, *i.e.*, a constraint variable stores its home space in its head.

Trailing Changes to Global Constraint Variables Changes to a global constraint variable have to be encapsulated. Hence, updating the domain of a global constraint variable by telling a basic constraint to it requires to trail the changes by taking a copy of the original domain. Since this operation is a very frequent operation, an efficient implementation has to avoid redundant trailing.

The standard technique is *time-stamping*, *i.e.*, a variable which is trailed is stamped with a virtual time. A variable is only trailed if the current time and the variable time is different. Since every space has its own virtual time, a variable is only trailed once per computation space. The straightforward scheme has an extra field that stores the complete time information [2]. Instead, a trailing scheme called *time-marking* [128] is used which requires just a truth value. Therefore, the implementation needs just a single bit (see Section 7.2.1 for details). The idea is to tag a variable as being time-stamped as soon as it is trailed to prevent further trailing. When moving on to a subordinated space, all tags of variables trailed in the child space are undone before creating a new trail frame. When coming back to this trail frame later on, all variables in the old trail frame are tagged trailed again.

Propagation Function Parameters of propagators can be global constraint variables. Only the propagation function of a propagator has to take care for global parameters. This is done by extending steps 1 and 3 in Figure 5.3 on page 39. Step 1 creates a working copy of the domain of a global and all domain updates are done on this copy. Step 3 is extended to update the actual domain of a global parameter with the working copy by trailing the changes.

5.3.3 Propagator Management Reconsidered

The introduction of spaces in the VM makes propagators situated, *i.e.*, a propagator is only run in its home space. The home space is stored in the head of the propagator.

Scheduling of Propagators A computation space encapsulates constraint propagation. This requires that only propagators in the current space sub-hierarchy are scheduled due

to changes of constraint variables in the current space. Hence, Figure 5.1 has to be updated accordingly to schedule only propagators in the current space sub-hierarchy.

Local Runnable Propagator Sets and Local Propagation Engine Moving computation from one space to another imposes an unnecessary overhead. It is desirable to perform as many propagator invocations as possible in an individual space without being interrupted by moving from one space to another. Hence, the global runnable propagator set (Figure 4.5 on page 32) is split into space-wise *local runnable propagator sets* (short LRP). Scheduling a monotonic propagator p, adds p to the LRP of p's home space. The union of all LRPs corresponds to the set of runnable monotonic propagators in Figure 4.7 on page 33. Monotonic propagators to be executed in a space S are stored in lrp_S (the LRP of S). The propagation engine for monotonic propagators in Section 5.2.3 is replaced by local propagation engines associated with computation spaces. A *local propagation engine* associated with S executes the propagators in lrp_S. As soon as lrp_S is not empty, an LRP-thread T_{lrp_S} is created and scheduled. Running T_{lrp_S} executes the local propagation engine of space S. This propagation engine can be preempted between two propagator function invocations to not interfere with the fairness of threads. After preemption, the corresponding LRP-thread T_{lrp_S} is immediately scheduled. If lrp_S is empty, the execution of the engine stops and the LRP-thread T_{lrp_S} is discarded. A new thread T'_{lrp_S} is created if a propagator is scheduled to in S and hence, added to lrp_S.

Running Non-monotonic Propagators Non-monotonic propagators are run in an order fixed by their *priority*; the earlier a propagator is created, the higher is its priority. Every space is assigned a *non-monotonic local runnable propagator set* (short NLRP). This is an ordered set of runnable non-monotonic propagators which dispenses propagators according to their priority. Scheduling a non-monotonic propagator p, adds p to the NLRP of p's home space. The union of all NLRPs represents the ordered set of runnable non-monotonic propagators in Figure 4.7 on page 33.

If lrp_S is empty the propagators in the NLRP $nlrp_S$ of a space S are run by local propagation engines for non-monotonic propagators. These propagation engines work identically to the local propagation engines for monotonic propagators except that the order of propagators is maintained. Running the propagators from $nlrp_S$ may turn other propagators in S `runnable` again and thus S may become instable again. A space S can only become stable if lrp_S and $nlrp_S$ are empty.

Locality of Parameters and Propagators The majority of parameters of propagators are local, *i.e.*, created in the same space as the propagators themselves. Running the constraint applications of the Mozart Oz test suite produces the figures in Table 5.2. These figures are similar to the figures given by Montelius in [97, Section 7.6] and show that 87 % of the parameters are local.

The observation that the vast majority of the parameters of a propagator are created in the same space as the propagator itself and the fact that propagators are typically added to event sets, suggests to enforce the invariant that event sets store only propagators that have the same home space as the parameter. This avoids the overhead of checking locality for every individual propagator once the locality of a parameter is checked.

number of all parameters	local parameters	global parameters
343.736	299.353	44.383

Table 5.2: Locality of propagator parameters.

Scheduling propagators amounts then to simply moving propagators of a event set to the local runnable propagator set of the current space without any locality check.

A propagator suspension on a global parameter is added to the parameter's suspension set instead of an event set. For propagator suspensions in a suspension set, locality has to be checked individually.

Stability For stability checking a space S, the VM needs (i) the parameters of the propagators in S (see Section 5.3.1) and (ii) the information if there are runnable propagators in S. Due to the requirement (i), every propagator is able to provide the VM with its parameters. The VM checks for propagators with global parameters. Requirement (ii) is covered by the mechanism for threads since all runnable propagators of S are represented by the LRP-thread T_{lrp_S}.

5.4 Discussion

The presented integration of propagation engines is fully orthogonal to search and garbage collection since both are based on copying. Hence, both issues need not to be considered for the integration. In contrast, hierarchies of computation spaces have a significant impact on the integration and require a considerable amount of design and implementation effort. Their trailing scheme for realizing encapsulation is not to be confused with the copy-based state restoration scheme of Mozart's search facilities.

An interesting scheduling scheme, called "layered propagation architecture", is proposed by Laburthe in [83]: instead of having only one runnable propagator set, there are different levels of expected propagation cost corresponding propagator sets. The set with lowest cost is run first and then the other sets according to the costs of the propagators. The presented model can "mimic" Laburthe's scheduling scheme for the case of two sets of runnable propagators by declaring expensive propagators as "non-monotonic" and thus, postponing their execution until all other propagators have reached a fixed-point.

The presented integration of propagation services is an conservative and orthogonal extension of the Mozart VM because it extends the functionality of the virtual machine via a clean interface without changing existing components of the VM.

Chapter 6

A Domain Solver Interface Architecture

This chapter presents the design of an interface for separating propagation services and domain solvers extending the architecture in Figure 4.1 on page 29 to the architecure in Figure 6.1.

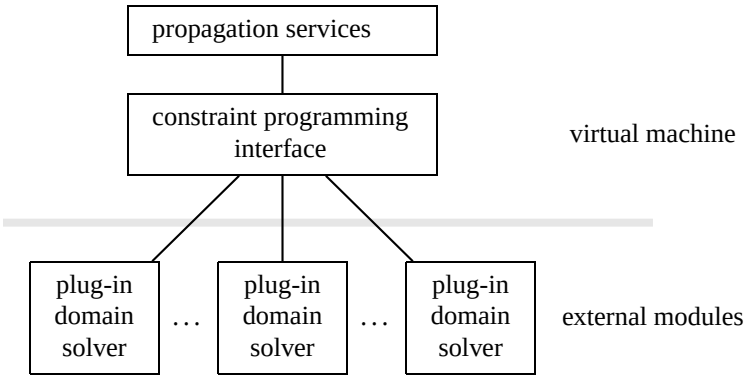


Figure 6.1: Architecture of a propagation engine with an interface between propagation services and domain solvers.

This makes it possible to connect different domain solvers by the interface to the propagation services. Domain solvers are now *external* modules plugged into the VM.

Additionally, this chapter proposes an interface for separating propagation functions and filters. Mozart's constraint propagator interface (Chapter 8) is based on this design.

6.1 Requirements

An interface between propagation services and domain solvers has to meet the following requirements:

Ease of Use The interface hides low-level issues like system particularities to enable the user to concentrate on issues like filtering and propagation techniques. On one hand,

the provided abstractions make it possible to solve common implementation tasks with minimal effort. On the other hand, the implementation of non-straightforward optimizations is possible too.

Expressiveness The interface is flexible enough to support state-of-the-art constraint solving techniques.

Extensibility The interface provides abstractions to make the complete implementation of new domain solvers possible. Furthermore, it is extensible to meet the requirements of future developments (for example, see Section 14.4 on the implementation of first-class constraints).

Minimality The interface provides only functionalities that cannot be obtained otherwise by the virtual machine. This makes the implementation simple and maintainable. This is because programmers tend to keep just a limited set of functions in mind and program their own extensions on top of them.

Compatibility and System-Independence The interface model can be easily implemented for various propagator-based engines. Filter for the propagation functions of propagators can be (re-)used without the need to change any of their code as demonstrated in the constraint library FIGARO [111].

Efficiency The interface services used by domain solvers have to be efficient.

6.2 Constraint Variables

This section discusses descriptions of new domain solvers and the creation of constraint variables.

Description of a Domain Solver (Figure 6.2) A domain solver $\mathcal{DS}$ is described by its type ($\texttt{type}_{\mathcal{DS}}$), its constraint domain ($\texttt{domain}_{\mathcal{DS}}$), and possible events ($\texttt{events}_{\mathcal{DS}}$). This information is collected in a *domain solver description*.

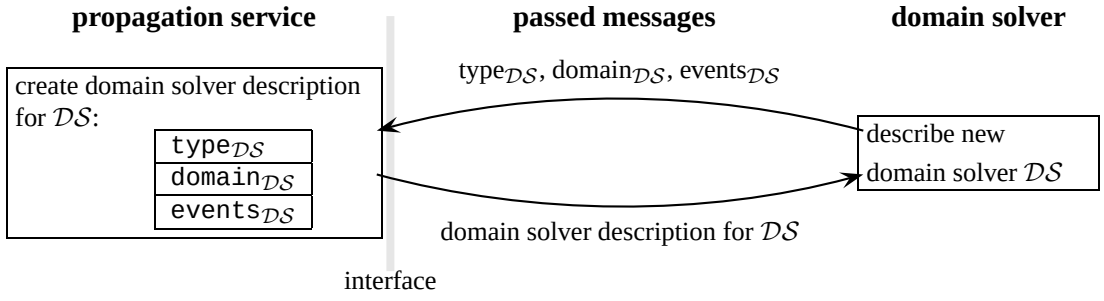


Figure 6.2: Description of a domain solver.

The type assigns a constraint variable to a certain domain solver. The constraint domain determines the "universe" of a domain solver. The events are needed for scheduling propagators.

Creating a Constraint Variable (Figure 6.3) Creating a new constraint variable needs

a domain solver description to be able to initialize the new variable with the type, an constraint domain, and a set of event lists.

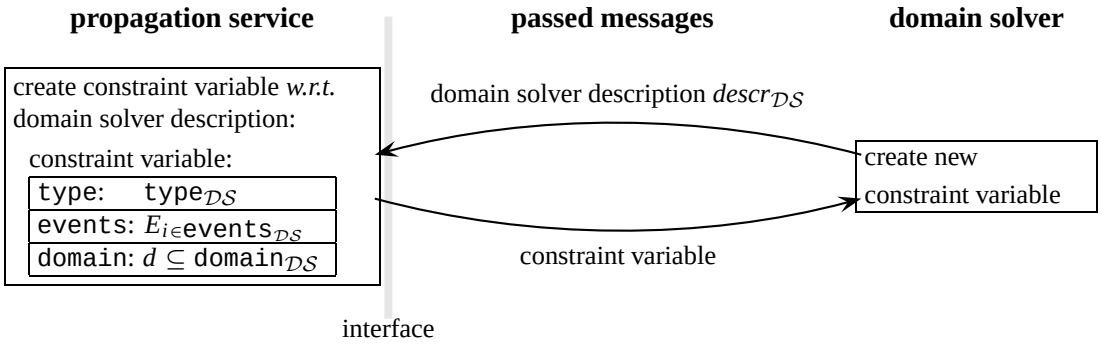


Figure 6.3: Creating a constraint variable.

The domain solver passes a domain solver description to the propagation service and obtains the newly created constraint variable.

6.3 Managing Propagators

This section identifies the operations on propagators to be made available by an interface.

Propagator Scheduling (Figure 6.4) Propagator scheduling needs to distinguish between monotonic and non-monotonic propagators. In case of the later ones, the priority has to be obtained additionally.

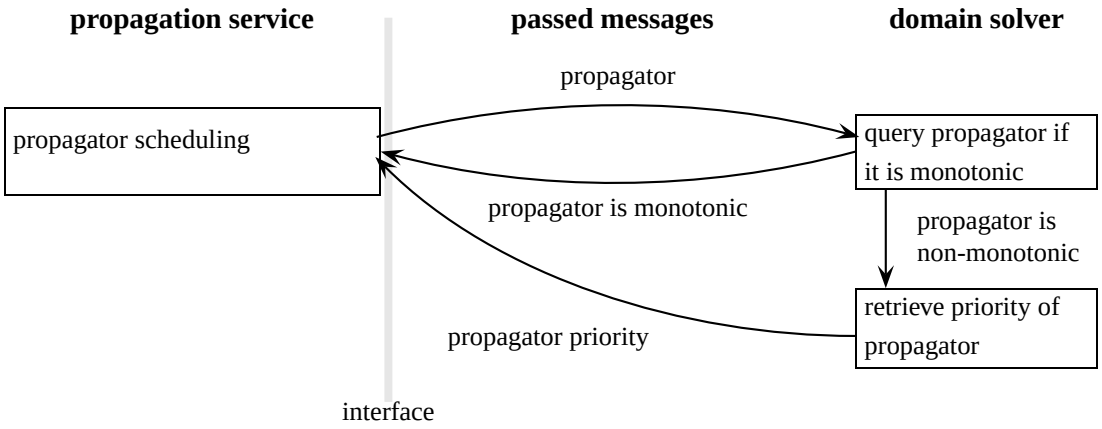


Figure 6.4: Propagator scheduling.

Stability Check (Figure 6.5) The stability check needs to analyze parameters of propagators which are situated in superordinated spaces. Hence, a propagator is requested

to provide its parameters to the propagation service. Note that this functionality is also essential for first-class constraints (see Section 14.2).

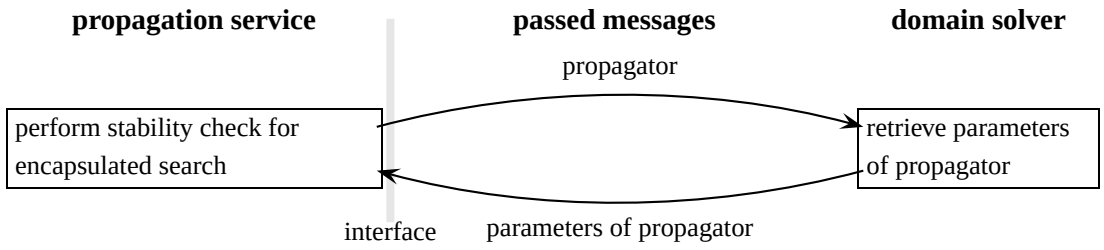


Figure 6.5: Stability check.

Creating a Propagator A propagator p_c for a constraint c is created and imposed on its parameters (Figure 6.6) if the basic constraints of its parameters b_{params} entail the constraints expected by the constraint on its parameters b_{expected}. If this is not the case, it is signaled to the propagation services: `suspend` is signaled if b_{expected} is not yet entailed and `type-error` if b_{expected} is dis-entailed. Otherwise, the events for re-executing the propagator are determined, the propagator is created and passed along with the events to the propagation service.

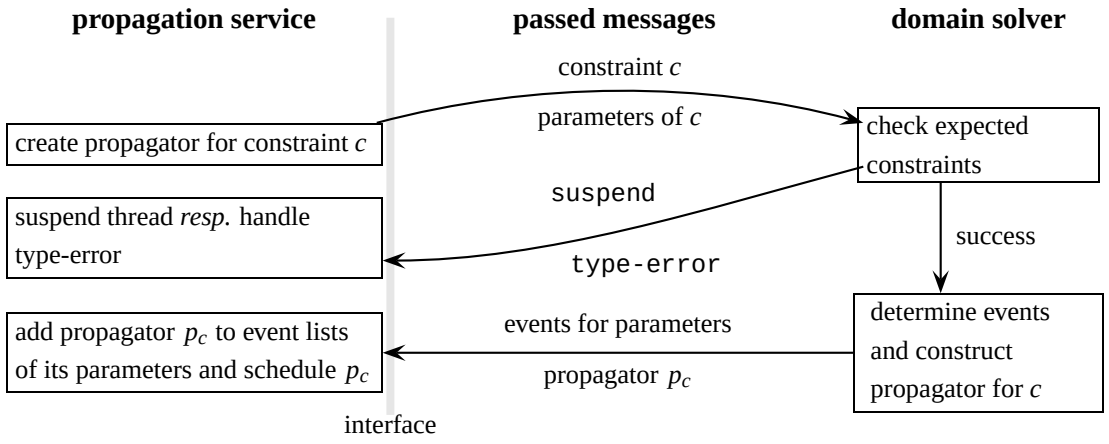


Figure 6.6: Creating a propagator.

6.4 Constraint Propagation

This section identifies the functionality required by a propagator to perform constraint propagation.

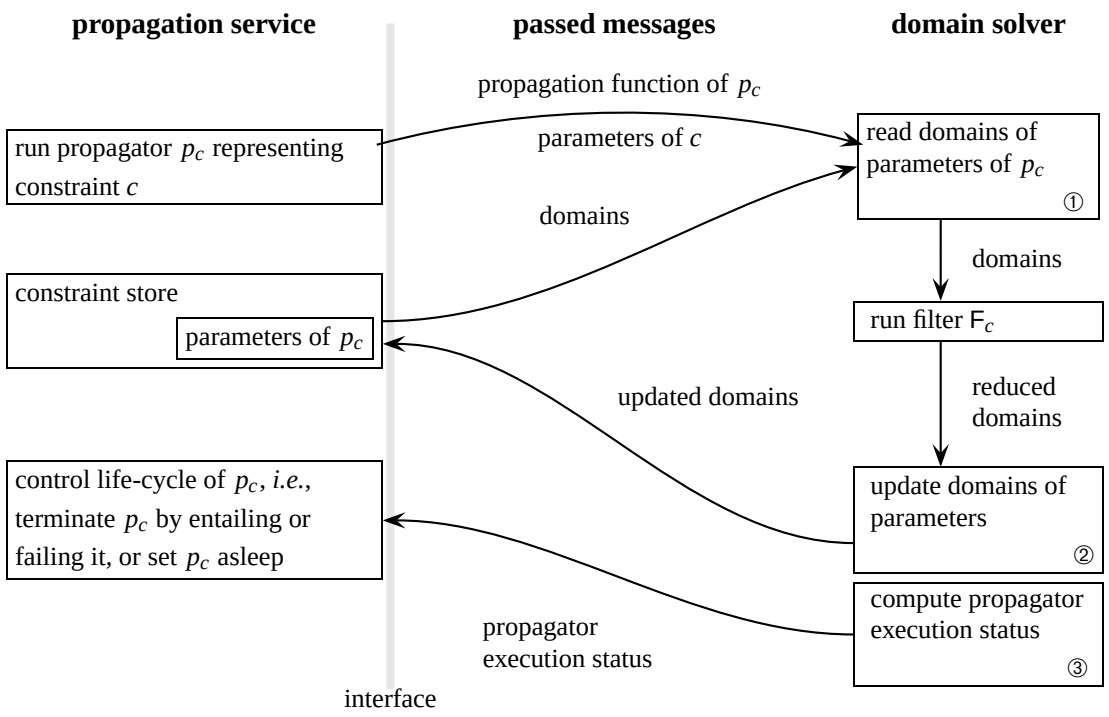


Figure 6.7: Running a propagation function.

Executing a Propagator (Figure 6.7) Executing a propagator means to run its propagation function. This is done either in the course of creating a propagator or if events occur on parameters of the propagator.

Domain reduction is performed by the filter of the propagator which runs on the domains of the parameters. Hence, the propagation function retrieves the domains of its parameters, invokes the filter, and finally updates the reduced domains of the parameters.

The propagator execution status is computed depending on whether the reduced constraints are entailed by the constraint store, inconsistent with the store, or none of it. The propagator execution status is passed to the propagation service which controls the life-cycle of the propagator according to Figure 2.4 on page 13.

Accessing Parameters (Figure 6.8) Accessing a propagator's parameter comprises to obtain access to the domain, to compute a profile, and when domain reduction (filtering) is finished, to update the parameter domain and to inform the propagation services which propagators have to be scheduled.

Propagator Creation by Propagator Function (Figure 6.9) While running the propagation function, the creation of a new propagator may be desired for example for optimizations. This newly created propagator $p_{c'}$ may either supplement or replace the existing propagator p_c (as in case of propagators for reified constrains). This is determined by the filter algorithm.

Propagator $p_{c'}$ is imposed on (a subset of) the parameters of p_c. The required func-

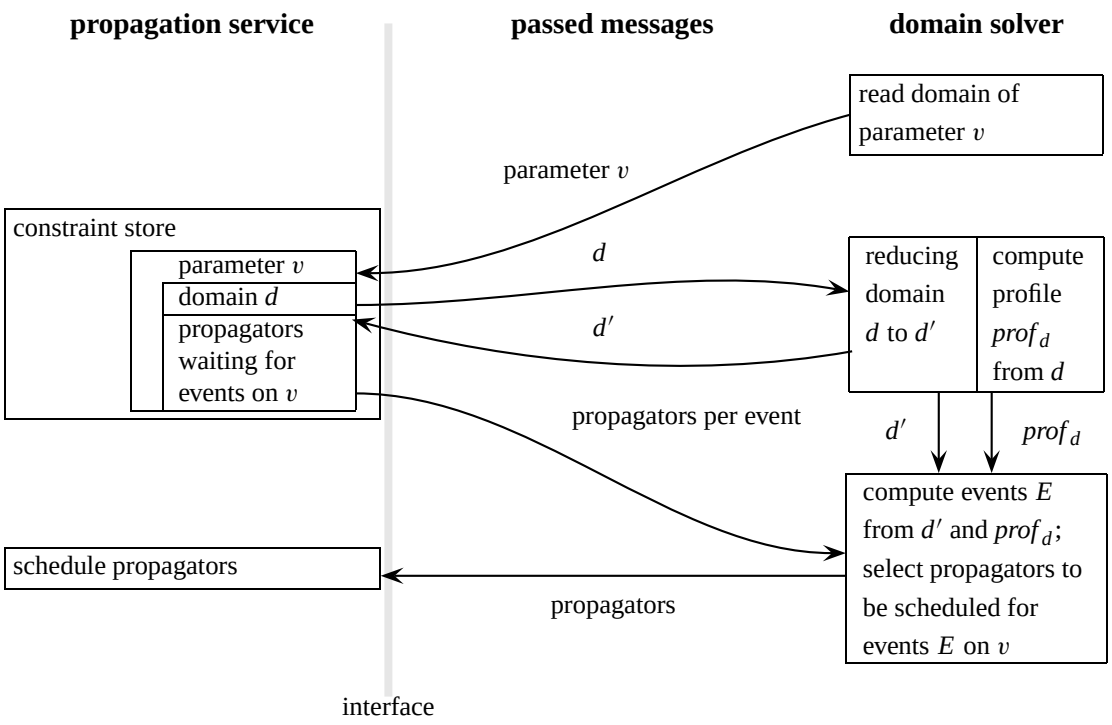


Figure 6.8: Accessing Parameter.

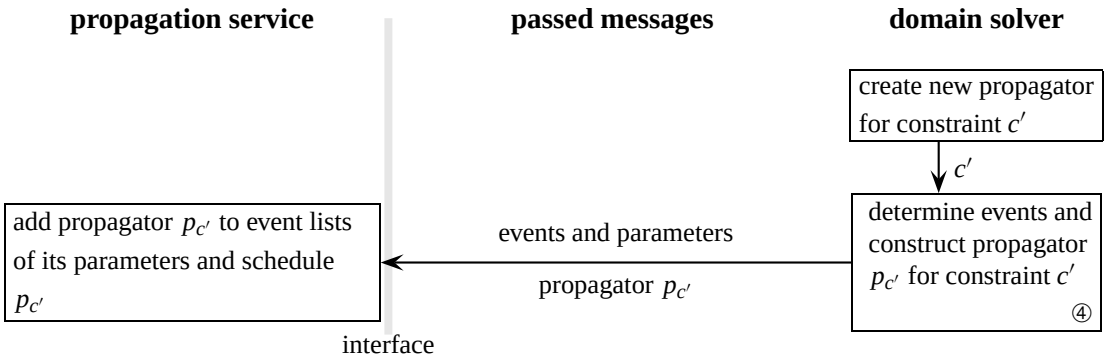


Figure 6.9: Propagator creation by propagator function.

tionality is a subset of the functionality for the creation of a propagator from scratch (Figure 6.6).

6.5 Separating Filters from Propagation Functions

A filter is independent of the host system and is desired to be shared between various implementations of domain solvers. Hence, it is desirable to isolate the services required by

a filter function and to define an interface that makes filter host system-independent and thus, straightforward to share. Figure 6.10 shows the interface between a propagation function and its filter.

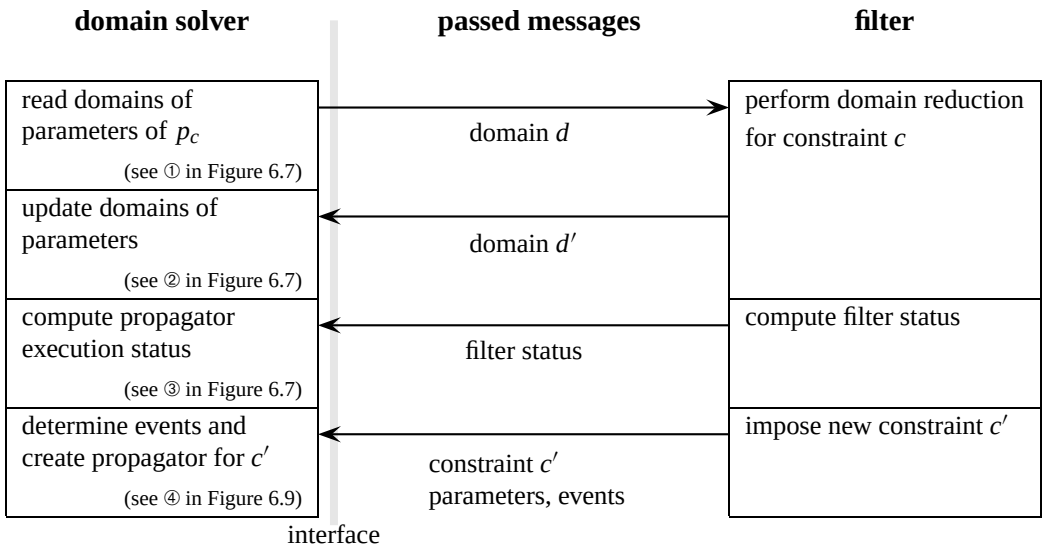


Figure 6.10: Separating the filter from the propagation function.

The filter interface relies on operations provided by the interface between the propagation service and the propagators (references ①–④ in Figure 6.10) and isolates such operations only relevant for connecting a filter function with a propagator.

The filter operates only on domains rather than on variables. The filter status is an abstract value which is translated into host system-dependent values. If the filter wants to impose an extra propagator, it passes the constraint to its host propagator which takes care of creating a new propagator.

6.6 Interface Abstractions

The identified services for an interface between propagation services and domain solvers have to be reflected by interface abstractions. The following abstractions are proposed for the implementation of a concrete interface:

Domain Solver Description It collects the type, domain, and the set of possible events.

Domain Representation It represents a domain attached to a constraint variable in the constraint store and provides abstractions for updating domains.

Constraint Profile It stores a constraint profile of the domain representation.

Set of Events It represents a set of events.

Creation of a Constraint Variable It creates a constraint variable according to a domain solver description.

Propagators It presents a propagator and supports the operations shown in Figure 6.4 and Figure 6.5.

Creation of a Propagator It joins the functionality shown in Figure 6.6.

Access Variable It joins the functionality shown in Figure 6.8.

Propagation Routine of a Propagator It joins the functionality shown in Figure 6.7 and Figure 6.9.

Filter It joins the functionality discussed in Section 6.5.

The implementation of the interface abstractions, depends (i) on the implementation language and (ii) on the host system. Chapter 8 presents an instance of this interface design for the Mozart in C++.

Chapter 7

Implementation Aspects of Propagation Services

This chapter discusses implementation aspects of the integration of propagation services into the Mozart VM according to the model presented in Chapter 5. This chapter is complemented by Chapter 8 which presents how domain solvers are connected to propagation services.

The code samples are given in C++ [109, 140] (i) to ease the reconstruction of the implementation of propagation services and (ii) to demonstrate that the model of propagation services can be implemented in a standard programming language in a concise, elegant and comprehensible way.

This chapter starts by describing briefly the interface classes of the Mozart VM (Section 7.1) and proceeds by discussing the integration of propagation services (Section 7.2).

7.1 Interfaces to the Services of the Virtual Machine

This section briefly presents the C++ class interfaces of the classes implementing the services of the Mozart VM described in Section 5.1. These classes are relevant since they are the interface between the propagation services and the rest of the Mozart VM and thus, the base for the integration of propagation services into VM. Mehl discusses the full implementation of the Mozart VM in [88].

Spaces The Mozart Oz VM implements spaces by the class `Space`:

```
class Space {
  bool isRoot(void);                    // check for root space
  static Space getCurrentSpace(void);  // get current space
};
```

Note `getCurrentSpace()` is static since there is exactly one current space in the VM.

Threads and the Scheduler A thread is defined by class `Thread`:

```
class Thread {
  Thread(Space home);   // construct thread
```

```
  Space getHome(void);  // get home space
};
```

A thread is passed on construction its home space by argument `home`. Threads are run by the VMs scheduler which is an instance of class `Scheduler`.

Variables The body of a variable is represented by an instance of class `Variable`:

```
class Variable {
public:
   Variable(Space home);        // construct variable
   Space getHome(void);         // get home space
   bool isLocal(Space current);
};
```

The current space is passed as argument `home` to the constructor `Variable()`. A variable is checked to have `current` as home space with `isLocal()`. Note the suspension set is implemented by a suspension list.

Values and Tagged References Nodes in the constraint store are implemented by *tagged references*. In case of a variable, a tagged reference implements a part of the head of the variable referring to an instance of class `Variable`. A tagged reference consists of a tag field, denoting the type of the referred value, and a value field, denoting the value itself.

A tagged reference is an instance of class `TR`:

```
class TR {
public:
  // value node for integers
  TR(int val);                      // constructor
  bool isInteger(void);             // type test
  int getInteger(void);             // access
  // variable node for variable head
  TR(Variable * var);               // constructor
  bool isVariable(void);            // type test
  Variable * getVariable(void);     // access
  // reference  node
  TR(TR * tr);                      // constructor
  bool isReference(void);           // type test
  TR deref(void);                   // access
};
```

There are three member functions for every type of value a tagged reference may refer to: a constructor, a type test, and an access function. A constructor initializes a new instance of `TR` and sets the tag field appropriately. Function `deref()` returns the last tagged reference of a reference chain.

Trail A trail is defined by class `Trail`.

```
class Trail {
public:
  void push(TR * var_loc, TR var);
```

```
};
```

It records changes to variables by its member function `push()` which stores the previous value `var` in conjunction with the location of `var`, namely `var_loc`. This makes it possible to restore the previous state and to produce a script for restoring the current state.

Foreign Functions Foreign functions are C-functions that obtain access to the constraint store via its arguments. The signature of a foreign function is:

```
typedef Status (*ForeignFun)(...);
```

The return value of a foreign function of type `Status` signals the virtual machine whether the application of the function was successful (`SUCCESS`), unsuccessful (`FAIL`), or suspended due to still unknown values (`SUSPEND`).

7.2 Constraint Propagation Services

This section discusses the implementation aspects of the integration of constraint propagation services into the Mozart VM. First, the integration of domain solver independent parts of constraint variables into the constraint store is explained (Section 7.2.1) followed by the integration of propagator heads (Section 7.2.2). Then, local propagation engines for executing propagators are integrated into the VM (Section 7.2.3). Finally, the implementation of a central routine on constraint variables is explained: the imposition of basic constraints on constraint variables (Section 7.2.4).

The implementation of propagator creator functions and filters for propagators is explained in Chapter 8 since these issues are subject to the constraint programming interface introduced there.

7.2.1 Constraint Variables

This section discusses the implementation of constraint variables. A constraint variable consists of a variable head (class `CtVariable`), a description of the constraint domain (class `CtDescr`), a representation of the domain (class `Ct`), and a set of event lists. Class `CtDescr` is the base class of all *domain solver descriptions*. It defines the type for the constraint variables (to what domain solver the variables belong to), the universe of the variables' domains and the possible events. Furthermore, it defines the name of the domain solver and the names of the events which are used, *e.g.*, for displaying domains by the system. There is exactly one instance of `CtDescr` for every domain solver. Class `Ct` is the base class for all domain representations. It provides the minimal functionality of a domain representation. Class `CtEvents` is the base class of all constraint events. It is a set of events which is computed either from (i) a constraint d' and its profile $prof_d$ or (ii) a constraint d' and its previous state d.

A concrete domain solver derives from the classes `CtDescr`, `Ct`, and `CtEvents` new classes and creates instances from them. Only class `CtVariable` remains unchanged for different domain solvers.

Representing Constraint Variables The structure of a constraint variable is shown Figure 7.1. It consists of the variable head and the extensions. The constraint variable head inherits from synchronized situated variables the fields `home_space` and `susp_list`.

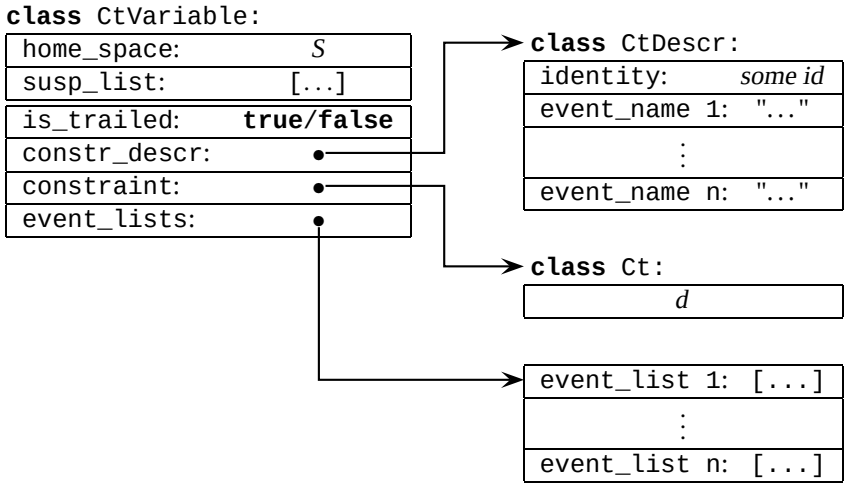


Figure 7.1: Structure of an instance of class `CtVariable`.

The additional fields are defined by the class `CtVariable`. Field `constraint` refers to the representation of the domain attached to the variable while field `event_lists` refers to the set of event lists. Field `constr_descr` refers to the domain solver description which gathers information to define and to identify the respective domain solver. Field `is_trailed` memorizes whether or not the variable is trailed. The implementation does not need extra memory since the flag is stored in an spare single bit.

```
class CtVariable : public Variable {
  CtVariable(Space space,              // constructor
             CtDescr descr, Ct constr);
  void updateConstraint(Ct constr);    // update constraint
  Ct getConstraint(void);              // access constraint
  void schedule(CtEvents events);      // schedule propagator
  void setTrailed(void);      // set variable 'trailed'
  void unsetTrailed(void);    // set variable 'untrailed'
  bool isTrailed(void);       // check if variable is 'trailed'
  CtVariable copyVar(void);   // copy variable
};
```

Program 7.1: Class definition of constraint variables.

The interface of class `CtVariable` is defined in Program 7.1. The constructor creates a constraint variable according to `descr` situated in `home` and with an initial domain

`constr`. Function `schedule()` schedules propagators in the event lists addressed by `events`. Note that `isLocal()` is inherited from class `Variable`.

Introducting the class `CtVariable` requires to extend class `TR`:

```
class TR { ...
  TR(CtVariable * cvar);
  bool isCtVariable(void);
  CtVariable * getCtVariable(void);
... };
```

Representing Domains Class `Ct` defines the minimal functionality of a class representing a domain (Program 7.2). A concrete representation of a domain is an instance of a class derived from `Ct`.

```
class Ct {
  Ct intersect(Ct constr); // compute intersection with 'constr'
  int getCard(void); // return cardinality
  TR getValue(void); // convert singleton domain to proper value
  CtEvents computeEvents(Ct constr); // events wrt. 'constr'
  bool isInDomain(TR val); // check if proper value is contained
};
```

Program 7.2: Class definition for domains.

Function `intersect()` returns the intersection of `constr` and the represented domain. The number of elements in a domain can be retrieved by `getCard()`. It is for the implementation of the propagation services only important to know whether a domain is empty, a singleton set or something else. If only one element is left in a domain, it can be retrieved by `getValue()`. Function `computeEvents()` supposes `constr` to be a previous state of the constraint and computes the events for the current constraint relative to `constr`. Whether a value `val` is in a domain or not, can be checked with `isInDomain()`.

7.2.2 Propagators

A propagator consists of a head and a body (Figure 4.3). The propagator head is defined by class `Propagator` (Program 7.3). This interface defines the functionality required by the propagation services. The body of a propagator is an instance of a concrete propagator class derived from class `Propagator`. The implementation of the propagation function is discussed in Section 8.5 while the propagator creation is considered in Section 8.4.

A propagator is situated and hence, it is initialized with its home space on construction. The initial execution state is *`running`*.

The stability check for propagators has to find out whether a propagator has a global variable as parameter or not. Hence, references to all propagators with global parameters are stored in their home spaces. During the stability check of a propagator's home

```
class Propagator {
  Propagator(Space home);               // construct propagator
  Space getHome(void);                  // get home space
  enum PropagatorState                  // execution states
      { RUNNABLE, RUNNING, SLEEPING, ENTAILED, FAILED };
  void setState(PropagatorState state); // set execution state
  PropagatorState getState(void);       // get execution state
  bool schedule(void);                  // schedule propagator
  TR getParameters(void);               // get parameters
  enum PropagatorStatus       // outcome of propagator function
      { SLEEP, FAIL, ENTAIL };
  PropagateStatus propagate(void);      // propagation function
  bool isNonMonotonic(void);    // propagator is non-monotonic?
  int getPriority(void);        // get priority
};
```

Program 7.3: Class definition for propagators.

space, the propagator's parameters can be retrieved by `getParameters()` and checked whether they refer to global variables or not.

The execution state of propagator is denoted by a value of the enumerable type `PropagatorState` correspond to the states in the boxes in Figure 2.4.

A propagator is scheduled by function `schedule()` which is discussed in detail in Section 7.2.3.

Constraint propagation of a propagator is performed by the propagation function. The result is returned as value of the enumerable type `PropagatorStatus` corresponding to the labels at the edges of Figure 2.4.

Adding Propagator Support to Variables The interface definition for synchronized and situated variables is extended by:

```
void Variable::addPropagator(Propagator * p);
```

which adds an entry for propagator `p` to the suspension list. Function

```
void Variable::schedule(Space current);
```

schedules all propagators of the suspension list which are situated in the space subhierarchy with root `current`.

The interface of class `CtVariable` is extended by

```
void CtVariable::addToEventList(CtEvents e, Propagator * p);
void CtVariable::addPropagator(CtEvents e,  Propagator * p) {
  if (getHome() == p->getHome())
    addToEventList(e, p);
  else
    Variable::addPropagator(p);
}
```

Function `addToEventList` adds propagator `p` to the event lists addressed by `e`.

According to the discussion of locality of propagator parameters in Section 5.3.3, a propagator `p` is only added to the event lists if the propagator has the same home space as its parameter. Otherwise, `p` is added to the suspension list inherited from class `Variable`.

Further, Table 5.1 on page 39 shows that 47 % of propagator creations result in immediate failure or entailment. Hence, propagators are only added to event *resp.* suspension lists if the initial run of the propagation function returns `SLEEP`.

7.2.3 Executing Propagators

A propagator is executed local to its home space by the *local propagation engine* of the home space. A local propagation engine is provided with computation time by an associated thread, the *local propagation engine thread*. Using a thread makes it possible to reuse the thread scheduling infrastructure of the VM.

The implementation of local propagation engines is discussed including their connection with threads and how propagators are passed to a propagation engine.

Local Propagation Engines

The implementation of the execution of monotonic and non-monotonic propagators is discussed in the following.

Executing Monotonic Propagators A local runnable propagator set is implemented by a stack of propagators. A propagator stack has the interface:

```
class PropagatorStack {
public:
  void push(Propagator * p);
  Propagator * pop(void);
  bool isEmpty(void);
};
```

The definition of the class `Space` is extended by a local propagation stack (`lps`) and the respective operations on it:

```
class Space { ...
  PropagatorStack lps;
  void addToLPS(Propagator * p);
  Status runLPS(Scheduler sched);
  ... };
```

Function `addToLPS()` adds `p` to the propagation stack `lps`. There is a thread (for short LPS-thread) associated with the non-empty `lps`. This thread runs `runLPS()` which executes the propagators in `lps`. The LPS-thread ceases to exist as soon as `lps` becomes empty. The LPS-thread represents the propagators in `lps` when the stability check determines whether or not there are runnable threads or propagators left.

The LPS-thread with home space `current` is created and immediately scheduled by:

```
void createRunnableThread(Space current, ForeignFun ffun);
```

The LPS-thread calls a foreign function that runs the local propagation stack. The definition of the foreign function is:

```
Status run_lps(void) {
  return Space::getCurrentSpace()->runLPS();
}
```

The LPS-thread is runnable as long as it exists and as soon as it is preempted, it is scheduled again. The thread can only be terminated by calling:

```
void terminateCurrentThread(void);
```

A propagator is submitted to a local propagation engine of a space *S* by calling the following member function of the instance of class `Space` representing *S*:

```
void Space::addToLPS(Propagator * p) {
  if (lps.isEmpty())
    createRunnableThread(this, run_lps);
  lps.push(p);
}
```

In case `lps` is empty, the LPS-thread is created to run the foreign function `run_lps` with the current space (**`this`**). Then propagator `p` is pushed onto `lps`.

The execution loop of the local propagation engine is:

```
Status Space::runLPS(Scheduler sched) {
  while (!lps.isEmpty() && !sched.preempt()) {
    Propagator * prop = lps.pop();
    switch (prop->propagate()) {
    case SLEEP:   prop->setState(sleeping);  break;
    case ENTAIL:  prop->setState(entailed);  break;
    case FAIL:    prop->setState(failed);    return FAIL;
    }
  }
  if (lps.isEmpty())
    terminateCurrentThread();
  return SUCCEED;
}
```

The execution loop iterates as long as `lps` is not empty and the scheduler does not preempt the currently running thread. This can be checked by:

```
bool Scheduler::preempt(void);
```

which returns **`true`** if the current thread is preempted. The body of the loop retrieves the next propagator from `lps` and runs its propagation function `propagate()`. The return value is evaluated by a **`switch`**-statement and sets the execution state of the propagator accordingly. In case a propagators fails, the function returns `FAIL` which is passed to the VM as return value of the foreign function. After leaving the loop, the LPS-thread is terminated in case `lps` is empty. Finally, `runLPS` returns `SUCCEED`, indicating that running the local propagation engine did not produce a failure.

Executing Non-monotonic Propagators Non-monotonic local propagator sets are implemented by *priority queues*. This is needed to ensure an execution order of non-monotonic propagators according to their priority. Propagators enqueued with the highest priority are dequeued first. The interface for a priority queue is:

```
class PriorityQueue {
  void enqueue(Propagator * p, int priority);
  Propagator * dequeue(void);
  bool isEmpty(void);
};
```

Class `Space` is extended by:

```
class Space { ...
  PriorityQueue nmlpq;
  void runNMLPQ(void);
  ... };
```

The purpose of `nmlpq` is to store runnable non-monotonic propagators until stability is checked in their order of creation. As part of the stability check, function `runNMLPQ()` is called which simply transfers the non-monotonic propagators to `lps`.

```
void Space::runNMLPQ(void) {
  while (!nmlpq.isEmpty())
    addToLPS(nmlpq.dequeue());
}
```

In case `nmlpq` is empty, the stability check proceeds. Otherwise, the space is unstable and the stability check fails immediately. Thus, the treatment of non-monotonic propagators is an orthogonal extension of the local propagation engine.

Scheduling a Propagator

A propagator is scheduled by calling either `CtVariable::schedule()` or `Variable::schedule()`. These functions traverse the suspension *resp.* event lists and apply

```
bool Propagator::schedule(void) {
  if (getState() == runnable)
    return false;
  setState(runnable);
  if (isNonMonotonic())
    getHome()->nmlpq->enqueue(this, getPriority());
  else
    getHome()->addToLPS(this);
  return true;
}
```

on the propagators to be scheduled. In case of event lists, this function is applied to all propagator suspensions in the respective event lists. For suspension lists, the current space must be on the path from the home space of the propagator to the root space.

The function returns **`true`** if the propagator is scheduled by this invocation. In case the propagator is already scheduled, this function returns **`false`**. If this is not the case, the propagator is set scheduled and added either to the local propagation stack or priority queue of its home space.

7.2.4 Telling Basic Constraints to Constraint Variables

Telling constraints to constraint variables is a key operation and for example needed by the finite domain operator *x*`::`*setdescr* (shown in Figure 3.3 on page 18).

The operation is implemented by function `imposeConstraint()` and uses two auxiliary functions which trail their changes on global variables. The first function `bind()` (Program 7.4) binds a variable to a value and trails the variable if it is global.

```
void bind(Trail trail, Space current, TR * tr_var, TR tr_val) {
  Variable * var = tr_var->getVariable();  // access variable
  if (! var->isLocal(current))    // trail ...
    trail.push(tr_var, *tr_var);  // ... global variable
  *tr_var = tr_val;               // bind 'tr_var' to 'tr_val'
}
```

Program 7.4: Binding a variable.

The second function `constraintCtVariable()` (Program 7.5) updates the domain of a constraint variable and trails the variable in case it is global. None of the auxiliary functions schedule propagators.

```
void constrainCtVariable(Trail trail, Space current,
                         CtVariable cvar, Ct constr) {
  if (! cvar.isLocal(current) &&      // trail only global ...
      ! cvar.isTrailed()) {           // ... untrailed variable
    trail.push(var, cvar.copyVar());  // store copy of variable
    cvar.setTrailed();                // set variable trailed
  }
  cvar.updateConstraint(constr);      // update constraint
}
```

Program 7.5: Constraining a constraint variable.

Function `imposeConstraint()` (Program 7.6) treats three cases:

1. The passed variable is not a constraint variable. The propagators in the suspension list are scheduled. The variable is bound to a newly created variable with `bind()`.
2. The passed variable is a constraint variable. The intersection of the domain of the variable and the passed domain `constr` is computed. If it is empty, the function

```
Status imposeConstraint(Trail trail, Space current,
                        TR * tr_var, Ct constr, CtDescr descr) {
  if (tr_var->isVariable()) {       // not a constraint variable
    tr_var->getVariable()->schedule(current);
    bind(trail, current, tr_var,
         TR(new CtVariable(current, descr, constr)));
  } else if (tr_var->isCtVariable()) {  // a constraint variable
    CtVariable cvar = tr_var->getCtVariable()
    Ct inter = constr.intersect(cvar.getConstraint());
    if (inter.getCard() == 0)
     return FAIL;
    cvar.schedule(inter.computeEvents(cvar.getConstraint()));
    constrainCtVariable(trail, current, cvar, inter);
  } else if (!constr.isInDomain(*tr_var))      // a proper value
      return FAIL;
  return SUCCEED;
}
```

Program 7.6: Imposing a basic constraints on an individual variable.

fails. Otherwise, compute the resulting events and schedule the sleeping propagators. Finally, update the domain of the variable with the intersection by `constrainCtVariable()`.

3. The passed variable denotes a proper value. It tests if the value is in the domain of the variable and if not, the function fails.

7.3 Discussion

A different approach to implement non-basic constraints are so-called *indexicals* (see [145]). This approach is used in many constraint solvers for WAM-based Prolog systems [149, 3] and was first presented by Codognet and Diaz in [32]. An indexical x in r constrains the variable x with the range r where r is computed from the current state of other variables. An indexical is re-executed if a variable in r is changed. Non-basic constraints expressed in terms of indexicals lead typically for more complex constraints (*e.g.*, n-ary linear equations [21, 32]) to a large number of indexicals. Hence, complex constraints are typically realized by library calls. The advantage of indexicals is that the propagation tasks are more fine-grain while the implementation of sophisticated filter is not well supported.

Zhou proposes in [151] to extend indexicals to *delay-clauses*. Additionally to head and body, a delay clause has a condition and a specification of events. This makes it possible to express sophisticated filter.

The implementation of constraint variables follows the line of *attributed variables* as used by Holzbaur to extend Prolog with constraints [67]. An attributed variable is a logic

variable annotated with attributes. An attribute can be for example a suspension list or a constraint domain. ECL^iPS^e uses attributed variable directly as a means to implement new domain solvers [76, 92].

The discussed implementation is optimized for propagation on local variables since global variables are only 13%. Furthermore, encapsulation is implemented such that operations on variable domains need not to be aware of trailing.

The VM of Mozart Oz represents finite domain and finite integer set variables more compact than shown in Figure 7.1. All components are directly stored in the variables avoiding the overhead of indirect access via references.

Mozart Oz uses a copying garbage collector and a copy-based search scheme. Copying support is orthogonal to the presented implementation of the propagation services and can be implemented by adding copy functionality to the respective class definitions.

Chapter 8

The Constraint Propagator Interface of Mozart

This chapter presents the *constraint propagator interface* of Mozart (short CPI) which is a C++ interface providing the substrate for building plug-in domain solvers. The interface connects domain solvers with the propagation services of the Mozart VM (Chapter 7) leading to propagation engines according to the achictecture shown in Figure 6.1 on page 45. Plug-in domain solvers connected by the CPI can cooperate via shared constraint variables with each other.

The CPI is an instance of the interface design presented in Chapter 6. The provided interface abstractions correspond to the abstractions proposed in Section 6.6.

This chapter addresses implementation issues of domain solvers as the domain-dependent extensions of the constraint store (Section 8.2), the definition of propagator bodies (Section 8.3), the creation of propagators including creator functions (Section 8.4), and the definition of propagation functions including the separation of filters (Section 8.5).

The application of the CPI is illustrated by implementing a plug-in finite domain solver including a propagator for the constraint $x \leq y + c$. This solver is used to analyze the computational cost imposed by various interfaces in Section 10.3 and can be obtained from [105].

8.1 Engineering the Concrete Interface

This section discusses design decisions to be made for engineering the CPI (Section 8.1.1) and provides an overview over interface abstractions which are explained in detail in the rest of this chapter (Section 8.1.2).

8.1.1 Design Decisions

Engineering an interface requires a couple of design decisions. First of all, the CPI is a C++-interface, *i.e.*, the interface abstractions are C++-classes. That makes it possible

to provide functionality but also to *insist* on functionality to be provided. On one hand, functionality is provided by interface classes and the user is free to derive new classes from interface classes. For more flexibility, *virtual member functions* are used, which enable dynamic binding of member functions. This makes it possible to control any concrete instance of a class only by having a pointer to it and thus, separating completely instances created via the CPI from the virtual machine. On the other hand, it can be enforced at compile-time that a certain member function is defined. This is achieved by C++'s *abstract base classes* which provide only the declaration, *i.e.*, only the type signature, of certain virtual functions but not their definition. Such a member function is called *pure virtual member function* and is annotated with "`=0`" after its argument list. A C++-compiler rejects every instance construction of a class that contains a pure virtual member function [109, 140].

8.1.2 Overview over the Interface Abstractions

The CPI makes it possible to implement new domain solvers, to extend existing domain solvers by new propagators, and to implement filters such that they can be shared by different host systems. The corresponding abstractions to fulfill these tasks are shown in Figure 8.1. The boxes denote components to be provided by a domain solver and the edges describe the mutual use of the components within a domain solver *resp.* the domain solver and the VM. As a convention, the names of CPI-abstractions begin with "`OZ_`".

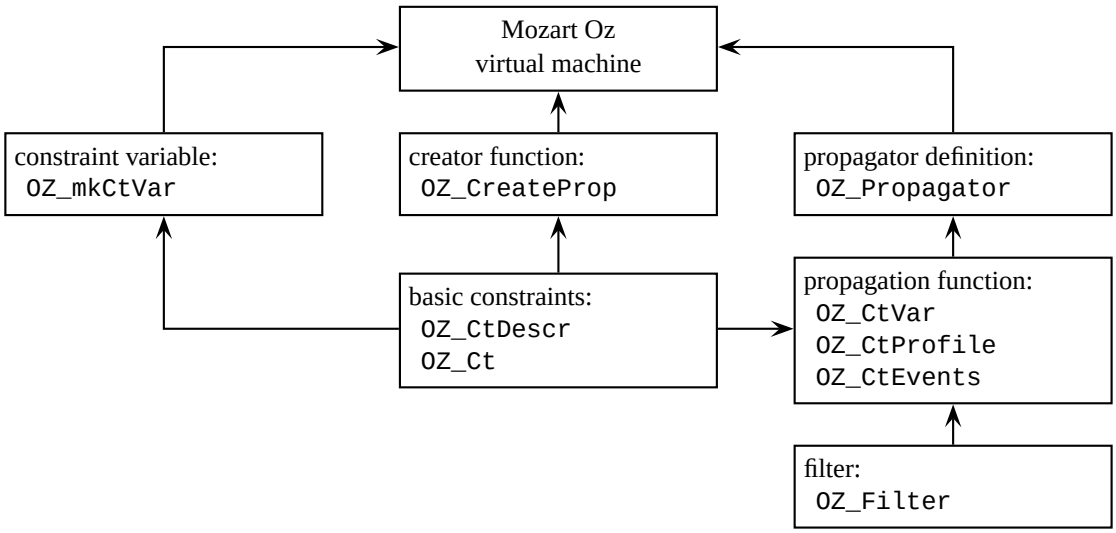


Figure 8.1: Overview of CPI-abstractions. Edges denote usage of abstractions by other blocks.

Constraint Variable and Basic Constraints A constraint variable is created by applying the function `OZ_mkCtVar()`. To create a constraint variable, a domain solver description is required which is an instance of a class derived from class `OZ_CtDescr`.

This class corresponds to class `CtDescr` in Section 7.2.1. Further, classes derived from class `OZ_Ct` represent basic constraints attached to constraint variables. Class `OZ_Ct` is an interface to class `Ct` shown in Program 7.2 on page 57.

Propagator Definition A propagator is defined by a class derived from class `OZ_Propagator`. This class is an interface to class `Propagator` shown in Program 7.3 on page 58.

Creator Function The propagator creator function is implemented as foreign function (Section 7.1). Checking for expected constraints at the parameters and imposing the propagator on its parameters is done by an instance of class `OZ_CreateProp`.

Propagation Function The propagation function obtains access to the basic constraints of the propagator's parameters by instances of class `OZ_CtVar`. Further, instances of this class in conjunction with instances of the classes `OZ_CtProfile` and `OZ_CtEvents` schedules sleeping propagators which wait for events on the parameters. Class `OZ_CtEvents` is an interface to class `CtEvents` introduced in Section 7.2.1.

Filter The CPI provides an interface to filters by the class `OZ_Filter`.

The CPI-abstractions discussed correspond to interface abstractions proposed in Section 6.6. This chapter does not explain every CPI-abstraction in detail. A detailed explanation of all interface abstractions can be found in [103]. Instead, the chapter focuses on the concepts behind the abstractions.

8.2 Constraint Variables, Profiles and Events

This section demonstrates how classes for a concrete domain solver are derived from the CPI-classes `OZ_Ct`, `OZ_CtDescr`, and `OZ_CtEvents`. The newly defined classes are part of a finite domain solver and are used in the following sections to implement a finite domain propagator including its creator function.

A Finite Domain Constraint A finite domain constraint (for short *domain*) is a finite set of integers. It is an instance of class `FD`.

```
class FD : public OZ_Ct { ... };
```

This class is derived from `OZ_Ct`. Class `FD` represents a finite set of integers in three different ways depending on its concrete value[1]: As long as the set denotes a range $[l, \ldots, u]$ it is represented only by the lower bound l and upper bound u. As soon as an individual element $e : l < e < u$ is removed the set is represented either by a bit vector (if $e < \mathit{threshold}$) or a list of intervals (if $e \geq \mathit{threshold}$). The threshold can be configured when starting up the system.

Finite Domain Description The constraint variables of the finite domain solver are identified by a single instance of class `FDDescr`. There is only one instance of a description class per domain solver.

[1] This representation is taken from the built-in finite domain solver of Mozart.

```
class FDDescr : public OZ_CtDescr {
  static int id;
public:
  virtual int getId(void)       // obtain identity
    { return id; }
  virtual char * getName(void)  // obtain name of domain solver
    { return "finite domain integer "; }
  virtual int getNoEvents(void) // number of possible events
     { return 3; }
  virtual char ** getEventNames(void) { // obtain event names
    static char * n[3] = { "value", "bounds", "domain"};
    return n;
  }
};
```

A domain solver is identified by a unique integer which is obtained by an application of the function `OZ_getUniqueId()`:

```
FDDescr::id = OZ_getUniqueId();
```

The finite domain solver is represented by an instance of class `FDDescr`:

```
FDDescr fd_descr;
```

Events There are three possible events defined in the finite domain description: an element is removed from a domain (event domain), at least one bound of a domain is narrowed (event bounds) and a domain is determined to a singleton value (event value).

A set of events is an instance of a class derived from the CPI-class `OZ_CtEvents`.

```
class FDEvents : public OZ_CtEvents {
public:
  FDEvents(void) { init(); }  // construct empty set of events
  void addValue(void)  { setEvent(0); }
  void addBounds(void) { setEvent(1); }
  void addDomain(void) { setEvent(2); }
  static OZ_CtEvents value(void);
  static OZ_CtEvents bounds(void);
  static OZ_CtEvents domain(void);
};
```

It is a set of maximum 32 elements and the integers assigned to the elements (see applications of `setEvent()` which are defined by `OZ_CtEvents`) correspond to the indices of the array of names returned by `FDDescr::getEventNames()`. The static functions `value()`, `bounds()`, and `domain()` denote sets of events for exactly the corresponding events. These functions are used in Section 8.4 for determining the events required the schedule a sleeping propagator.

Constraint Profile A constraint profile stores sufficient information of a previous state of a domain needed to compute events from a later state of the same domain. The advantage of a profile is that it takes typically much less memory than the complete domain.

All constraint profiles are derived from CPI-class `CtProfile`. The profile for finite domains is defined by class `FDProfile`:

```
class FDProfile : public OZ_CtProfile {
  int width, card;
  void init(OZ_Ct * c) {
    card  = ((FD *) c)->getCard();
    width = ((FD *) c)->getWidth();
  }
public:
  FDProfile(OZ_Ct * c) { init(c); }
  int getWidth(void) { return width; }
  int getCard(void) { return card; }
};
```

To compute finite domain events for a domain, it is sufficient to store the number of elements of the previous state of the domain (field `card`) and the difference of the largest and smallest elements (field `width`). Class `FD` provides the functions `getCard()` and `getWidth()` to obtain the current cardinality and width of a domain.

Computing Events Class `OZ_Ct` requires to define extra functions for obtaining the current state of a domain as constraint profile and for computing from the current state of a domain and a previously taken profile events. In the following, these functions for the finite domain solver are defined.

The current state of a domain is obtained by the function `FD::getProfile()` which returns a constraint profile as instance of `OZ_CtProfile`. The implementation of this function uses function `FDProfile::init()` (see above).

Finite domain events are computed by function `FD::computeEvents()` from the current state of a domain and a previously taken profile (argument `profile`):

```
OZ_CtEvents FD::computeEvents(OZ_CtProfile * profile) {
  FDEvents events;
  FDProfile * fdp = (FDProfile *) profile;
  if (getCard() == 1 && fdp->getCard() > 1) { // 'value' event
    events.addValue(); events.addBounds(); events.addDomain();
  } else if (getWidth() < fdp->getWidth()) {  // 'bounds' event
    events.addBounds(); events.addDomain();
  } else if (getCard() < fdp->getCard())      // 'domain' event
    events.addDomain();
  return events;
}
```

This function compares the cardinality and width previously stored in a profile with current values of a domain and sets the corresponding events in the events set `events`. Note that a value event includes also bounds and domain events while a bounds event includes a domain event.

Creating Constraint Variables There is no class representing a constraint variable in the CPI. Instead, there is function `OZ_mkCtVar()` for creating a new constraint variable

from a given domain description `descr` and an initial constraint `constr` and binding it to `var`.

```
OZ_Return
OZ_mkCtVar (OZ_Term var, OZ_Ct * constr, OZ_CtDescr * descr);
```

This function is an interface to function `imposeConstraint()` presented in Program 7.6 on page 63.

The return type `OZ_Return` of `OZ_mkCtVar()` corresponds to type `Status` while argument type `OZ_Term` denoting a tagged reference corresponds to class `TR`[2] (see Section 7.1 for details on both, `Status` and `TR`).

8.3 Propagator Definition

A concrete propagator is defined by a class derived from class `OZ_Propagator`. This class is an interface to class `Propagator` shown in Program 7.3. Additionally, class `OZ_Propagator` provides extra functionality (for example for execution profiling and memory management) which is explained in detail in [103].

A finite domain propagator for the $x \leq y + c$-constraint is defined. This propagator stores in its private state references to its parameters x and y and the value of c (`_c`). The references to x and y, namely `_x` and `_y`, are tagged references of type `OZ_Term`.

```
class Leqoff : public OZ_Propagator {
  OZ_Term _x, _y;
  int _c;
public:
  Leqoff(OZ_Term x, OZ_Term y,  OZ_Term c)
    : _x(x), _y(y), _c(OZ_intToC(c)) { }
  virtual OZ_Return propagate(void);
  ...
};
```

The constructor receives the parameters and initializes the corresponding fields. Function `OZ_intToC()` converts a tagged reference representing an integer to an C-integer. The constructor is used for creating an instance of the propagator in Section 8.4. The definition of function `propagate()` is discussed in detail in Section 8.5.

8.4 Propagator Creation

A propagator is created in three steps according to Figure 5.2 on page 38. The domains of the parameters are checked to entail the expected constraint (step 1), the creation of the propagator (step 2), and the imposition of the propagator (step 3). Step 2 is done by allocating memory for the propagator and calling the constructor of the propagator. Step 1 and 3 are taken care of by the CPI-class `OZ_CreateProp` (Program 8.1).

[2] In Mozart, a tagged reference is represented by an integer value (`int`) where certain bits are used to represent the tag and the remaining bits represent the value.

```
class OZ_CreateProp {
  OZ_expect_t
  expectInt(OZ_Term val);        // expect an integer
  OZ_expect_t
  expectCtVar(OZ_Term v,         // expect a constraint variable
              OZ_CtDescr * descr,
              OZ_CtEvents events);
  OZ_expect_t
  expectVector(OZ_Term v,        // expect vector
               OZ_CreatePropMeth f);
  bool isSuspending(OZ_expect_t r);
  bool isFailing(OZ_expect_t r);
  OZ_Return suspend(void);       // suspend creator function
  OZ_Return fail(void);          // fail creator function
  OZ_Return impose(OZ_Propagator * prop); // impose 'prop'
};
```

Program 8.1: Interface of CPI-class `OZ_CreateProp`.

Expressing the Expected Constraint of a Parameter The expected constraint of a parameter is checked by an `expect`-function. Each `expect`-function corresponds to a certain expected constraint. For example, a parameter is expected to be an integer (`expectInt()`) or a constraint variable of a certain domain solver (`expectCtVar()`, `descr` determines the domain solver). Additionally, there are iterators for compound values as vectors: the function `expectVector()` expects the parameter `v` to be a vector of values specified by the passed expect function `f()`. For example,

```
expectVector(vec, expectInt)
```

expects `vec` to be a vector of integers.

Evaluating Constraints at Parameters The return type of an `expect`-function `OZ_expect_t` denotes the number of constraints present (*e.g.*, number of elements of a vector) and the number of elements entailed by the parameter. This makes it possible to express propagators which can be nested (see below). A value of type `OZ_expect_t` is tested to denote a none-entailed constraint by function `isSuspending()` and to denote an inconsistent constraint by function `isFailing()`. If none of both functions is true, the expected constraint is entailed and the propagator is imposed on its parameters (see below).

The creator function is left by calling `fail()` if the domain at the parameter is inconsistent with the expected constraint or by calling `suspend()` in case the parameter do not entail the expected constraints.

Expecting a Constraint Variable The function `expectCtVar()` expects a parameter to be a constraint variable and determines the events the propagator is scheduled on this parameter (argument `events`). Further, if the parameter is neither entailed nor failed and

the creator function is not left, the expected constraint[3] and the propagator is imposed. This behavior is used to implement propagators that can be nested, since nesting makes shared parameters inaccessible, and the creator function has to impose the most universal domain on the shared parameter.

Propagator Imposition The variables encountered in the course of checking expected constraints are stored in the instance of `OZ_CreateProp` together with their events. The propagator created by the **`new`**-operator defined by `OZ_Propagator` and the constructor of the propagator class (see example below).

The propagator is passed to function `impose()` and the propagation function of the propagator is initially run. If the function does not indicate that the propagator is entailed or failed, the propagator is imposed on its parameters by adding entries to the respective event lists.

Example This example presents the creator function for the $x \leq y + c$-propagator. A creator function is a foreign function as discussed in Section 7.1. The evaluation of the return values for the individual parameters is done by the macro `OZ_EXPECT`. A macro is used to be able to leave the creator function by **`return`**-statements. The parameter `O` is an instance of class `OZ_CreateProp` and parameter `F` an application of a member function of `OZ_CreateProp`. The outcome of the application of `F` is first tested to fail or then to suspend the creator function.

```
#define OZ_EXPECT(O, F)          \
  { OZ_expect_t r = O.F;         \
    if (O.isFailing(r))          \
      return O.fail();           \
    else if (O.isSuspending(r))  \
      return O.suspend(); }
```

Suppose `x`, `y`, and `c` denote the corresponding parameters, the following CPI-code checks the expected constraints at the parameters, determines the events, creates a new $x \leq y + c$-propagator and imposes the propagator on its parameters:

```
OZ_CreateProp cp;
OZ_EXPECT(cp, expectCtVar(x, fd_descr, FDEvents::bounds()));
OZ_EXPECT(cp, expectCtVar(y, fd_descr, FDEvents::bounds()));
OZ_EXPECT(cp, expectInt(c));
return cp.impose(new Leqoff(x, y, c));
```

8.5 Propagation Functions

This section discusses the implementation of propagation functions which proceeds in three steps: first, access variables which provide access to constraint variables in the constraint store from within a propagation function (Section 8.5.1) are discussed. Then, the filter function interface is considered and the implementation of a filter function

[3]The parameter is constrained to a constraint variable of the domain solver with the most general domain.

(Section 8.5.2) is presented, and eventually access variables and a filter function are joined to a propagation function (Section 8.5.3).

8.5.1 Access to Constraint Variables

Access variables implement steps 1 and 3 of Figure 5.3 on page 39 to make the execution of the filter function (step 2) possible.

The Interface An access variable is an instance of a class derived from the CPI-class `OZ_CtVar` (Program 8.2).

```
class OZ_CtVar {
  void read(OZ_Term p);      // read (effective propagation)
  void readEncap(OZ_Term p); // read (encapsulated propagation)
  bool leave(void);          // write domains to store
  void fail(void);           // fail access variable
protected:
  // handling domains, values, and profiles
  virtual OZ_Ct * ctGetConstraint(void) = 0;
  virtual void ctSetValue(OZ_Term val) = 0;
  virtual OZ_Ct * ctRefConstraint(OZ_Ct * c) = 0;
  virtual OZ_Ct * ctSaveConstraint(OZ_Ct * c) = 0;
  virtual void ctRestoreConstraint(void) = 0;
  virtual OZ_Ct * ctSaveEncapConstraint(OZ_Ct * c) = 0;
  virtual void ctSetConstraintProfile(void) = 0;
  virtual OZ_CtProfile * ctGetConstraintProfile(void) = 0;
};
```

Program 8.2: Interface of CPI-class `OZ_CtVar`.

Step 1 is implemented by `read()` and `readEncap()`. All changes to the domain of a constraint variable read in by `read()` are visible to the constraint store (effective propagation). This is in contrast to changes on variables read in by `readEncap()` which are invisible to the constraint store and hence providing for encapsulated propagation used by reified constraints.

Step 3 is implemented by `leave()` and `fail()`. While `fail()` is to be called for all access variables in case of failure, `leave()` writes domains to the connected constraint variable and schedules waiting propagators. The return value of `leave()` indicates whether the connected constraint variable is still undetermined (**`true`**) or not (**`false`**).

Defining an Access Variable Class

The functions `read()`, `readEncap()`, `leave()`, and `fail()` are implemented by a set of auxiliary functions. The definitions of these functions for finite domain constraints are discussed using the respective definitions from Section 8.2.

Figure 8.2 shows the structure of a finite domain access variable. The domain is only accessed via the field `domain`. Depending on the kind of constraint variable (local or global) and kind of propagation (effective or encapsulated), `domain` refers directly to the domain of the constraint variable in the store (②), to `copy` in case of a singleton or a global variable (① and ③), or to `encap` in case of an encapsulated variable (④). The references ①–④ are created by auxiliary functions which are explained in the list on page 75.

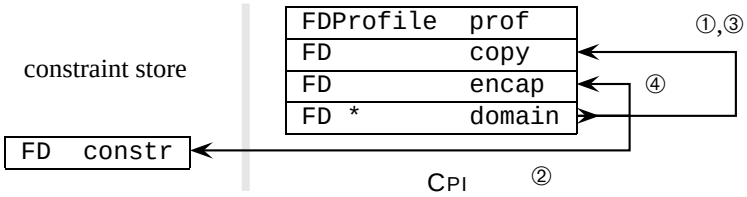


Figure 8.2: Structure of a finite domain access variable and .

Before the auxiliary functions are discussed, the class for the finite domain access variables is defined. This definition uses the classes `FD` (domain representation) and `FDProfile` (finite domain constraint profile) which are explained in Section 8.2.

```
class FDVar : public OZ_CtVar {
  FD * domain, copy, encap;
  FDProfile prof;
public:
  FDVar(OZ_Term t) : OZ_CtVar() { read(t); } // constructor
  FD &operator * (void) { return *domain; }
  FD * operator -> (void) { return domain; }
protected:
  // auxiliary functions go here
};
```

The field `domain` is a pointer to be directed to the appropriate domain representation when constructing an instance of this class. The operators `*` and `->` are used to either access the domain directly and to apply member functions, respectively. This follows the convention that an access variable behaves like a pointer to a domain representation.

Auxiliary Functions for Constraint Profiles The field `prof` stores the constraint profile needed to compute the events for scheduling propagators. The following auxiliary functions initialize *resp.* retrieve the constraint profile.

```
virtual void FDVar::ctSetConstraintProfile(void) {
  prof = domain->getProfile();
}
virtual OZ_CtProfile * FDVar::ctGetConstraintProfile(void) {
  return &prof;
}
```

Auxiliary Functions for Domains Function `ctGetConstraint()` is internally used to obtain access to the domains. It is defined as:

```
virtual OZ_Ct * FDVar::ctGetConstraint(void) { return domain; }
```

The remaining auxiliary functions are explained in correspondence to the references ①–④ in Figure 8.2 and defined for finite domain access variables.

① The parameter is a *singleton value*. Function `ctSetValue()` initializes `copy` to a singleton domain containing `val` and directing `domain` to the location of `copy`.

```
virtual void FDVar::ctSetValue(OZ_Term val) {
  copy.initDescr(val);
  domain = &copy;
}
```

The function `FD::initDescr()` initializes the domain according to the set of integers described by an Oz-value (see Figure 3.3 on page 18).

② The parameter is a *local* variable. The field `domain` is directed to the location of the constraint representation in the store (passed by `constr`) by function `ctRefConstraint()`.

```
virtual OZ_Ct * FDVar::ctRefConstraint(OZ_Ct * c) {
  return domain = (FD *) c;
}
```

③ The parameter is a *global* variable. The field `copy` is updated by function `ctSaveConstraint()` with the domain of the connected constraint variable. The current state of `copy` is written to the connected global constraint variable by function `constrainCtVariable()` (Program 7.5) when leaving the propagator.

```
virtual OZ_Ct * FDVar::ctSaveConstraint(OZ_Ct * c) {
  copy = * (FD *) c;
  return &copy;
}
```

④ The parameter is *encapsulated*, *i.e.*, constraint propagation is invisible to the constraint store. Hence, the field `encap` is updated by function `ctSaveEncapConstraint()` with the domain of the connected constraint variable. All constraint propagation is done on `encap`.

```
virtual OZ_Ct * FDVar::ctSaveEncapConstraint(OZ_Ct * c) {
  encap = * (FD *) c;
  return &encap;
}
```

In case a parameter is a *top-level* variable, it is treated like a local variable (②). Since the domain of a top-level variable must never become empty, the original domain is copied (*i.e.*, backed up) to the field `copy` and in case of a failure, written back to the store by `ctRestoreConstraint()`.

```
virtual void FDVar::ctRestoreConstraint(void) { *domain = copy; }
```

Note that there have to be separate fields for global and encapsulated parameters since both kinds of parameters can occur in a propagator at the same time (see Section 9.4.2 for more details).

8.5.2 Filter Interface

The filter sub-interface of the CPI provides a uniform way to communicate results and requests of a filter function to the invoking propagator function. Redefining the abstractions of the filter interface makes the application of existing filter functions in different host solvers straightforward avoiding to re-implement the complicated filter algorithms.

The interface defines class `OZ_Filter` which communicates results and requests of a filter function to the host solver.

The definition of class `OZ_Filter` is presented as an instance of the a template class `Filter` to emphasis the independence of the host solver. The interface of class `Filter` is shown in Program 8.3.

```
template <class AccVar, class AccVarIt, class Evts, class Prop>
class Filter {
public:
  // communicating filtering results
  Filter &fail(void);              // indicate failed filter
  Filter &entail(void);            // indicate entailed filter
  Filter &leave(int vars_left = 0);
  // propagator and parameter manipulation
  Filter &drop_parameter(AccVar &var);
  Filter &add_parameter(AccVar &var, Evts events);
  Filter &impose_propagator(Prop * prop);
  Filter &replace_propagator(Prop * prop);
};
```

Program 8.3: Interface of template class `Filter` (`AccVar` denotes an access variable, `AccVarIt` an access variable iterator, `Evts` a set of events, and `Prop` a propagator).

The class `OZ_Filter` is defined corresponding to the following type definition:

```
typedef Filter<OZ_CtVar, OZ_CtEvents,
               OZ_ParamIterator, OZ_Propagator> OZ_Filter;
```

A filter class for a different host solver can be defined by providing a type definition for the respective host solver. The following discussion refers to the CPI-instantiation `OZ_Filter` of class `Filter`.

The Actual Filter Function The following signature of a filter function `filter` is used for defining filter functions:

```
OZ_Filter &filter(OZ_Filter &service, ...);
```

An instance of `OZ_Filter` is passed as first argument and returned as return value to free the filter function entirely from the management of instances of class `OZ_Filter`. The ellipsis ("...") is a placeholder for the access variables to be passed. The member functions of class `Filter` (Program 8.3) have the return type `Filter` to make it possible to call them in a **`return`**-statement leaving a filter function with the proposed signature.

Communicating Filtering Results The outcome of running a filter function is communicated to the host system by applying `fail()` and `entail()` with the suggested meaning. Function `leave()` takes an integer denoting the number of undetermined parameters left to still entail the propagator running the filter. Otherwise, the host propagator is set to execution state *`sleeping`*.

Propagator and Parameter Manipulation A filter function can drop a parameter by function `drop_parameter()` and can add an extra parameter by `add_parameter()`. Further, a filter can impose a propagator (`impose_propagator()`) *resp.* replace its host propagator by another propagator (`replace_propagator()`).[4] The host solver is responsible for providing means to impose a propagator on a set of access variables similar to creator functions.

8.5.3 An Example of a Propagation Function

This section combines access variables and the filter function to define the propagation function for the $x \leq y + c$-constraint. First, the filter function is defined. Then, the definition of class `OZ_Filter` is extended by Mozart-specific functionality and eventually the propagation function of the $x \leq y + c$-propagator is defined.

Filter Function The propagation rules to be implement are:

$$\overline{x} \leq \underline{y} + c \longrightarrow \top \quad (8.1) \qquad x \leq \overline{y} + c \quad (8.3)$$

$$\underline{x} > \overline{y} + c \longrightarrow \bot \quad (8.2) \qquad \underline{x} - c \leq y \quad (8.4)$$

The lower (upper) bound of a finite domain variable x is denoted $\underline{x}$ ($\overline{x}$). Rule 8.1 detects entailment while rule (8.2) fires on failure. Constraint propagation on the upper bound of x is done by rule (8.3). The lower bound of y is constrained by rule (8.4). This rule subsumes the failure rule (8.2). It produces under the condition of the failure rule an empty domain of y which raises a failure.

The filter function is a template function parameterized over the types of the filter class (`FILTER`) and the finite domain access variable (`FDVAR`). Using templates has the advantage that filter implementations are independent from the names of classes used on a host solver and thus, can be used without any changes. Remember the convention, that an access variable behaves like a pointer to a finite domain representation `FD`. The operator `FD::operator >=` (`FD::operator <=`) constrains the lower (upper) bound of a

[4] Function `replace_propagator()` is typically used in propagators for reified constraints.

finite domain while function `FD::getMinElem()` (`FD::getMaxElem()`) obtains the minimum (maximum) element of a finite domain. The definition of the filter function is a direct implementation of the propagation rules.

```
template  <class FILTER, class FDVAR>
FILTER &filter_leqoff(FILTER &s, FDVAR &x, FDVAR &y, int c) {
  FailOnEmpty(*x <= (y->getMaxElem() + c));   // (8.3)
  FailOnEmpty(*y >= (x->getMinElem() - c));   // (8.4) and (8.2)
  if (x->getMaxElem() <= y->getMinElem() + c) // (8.1)
    return s.entail();
  return s.leave();
 failure:
  return s.fail();
}
```

Filter Functions for Propagators The filter interface in Program 8.3 defines only the functionality from the perspective of a filter function. A host solver needs additional facilities, as in case of Mozart, a constructor and a operator to realize the requests and to compute the return value for the propagation function.

```
class OZ_Filter { ...
  OZ_Filter(OZ_Propagator * prop,    // constructor
            OZ_ParamIterator * iter);
  OZ_Return operator ();  // realize requests and compute
                          // return value of propagation function
}
```

The constructor is passed the host propagator (`prop`) to be able to process requests as propagator replacement and an iterator over the access variables of parameters (`iter`).

```
class OZ_ParamIterator {
public:
  virtual OZ_Return leave(int vars_left = 0) = 0;
  virtual OZ_Return entail(void) = 0;
  virtual OZ_Return fail(void) = 0;
};
```

A parameter iterator is an instance of a class derived from `OZ_ParamIterator` and applies the corresponding functions of all the access variables it is initialized with. The propagator function for the $x \leq y + c$-propagator uses the parameter iterator `ParamIterator_V_V` taking care of two finite domain access variables. It provides a constructor and the functions `leave()`, `entail()` and `fail()`.

Propagation Function The propagation function `Leqoff::propagate()` is a combination of previously defined abstractions.

```
OZ_Return Leqoff::propagate(void) {
  FDVar x(_x), y(_y);
  ParamIterator_V_V params(x, y);
  OZ_Filter s(this, &params);
  return filter_leqoff(s, x, y, _c)();
}
```

First, the access variables for the parameters *x* and *y* are initialized with the tagged references `_x` and `_y` stored in the propagator. Then, the parameter iterator `params` is constructed. It is passed to the constructor of the `OZ_Filter` instance which connects the propagator with filter function `filter_leqoff()`. Finally, the filter function is called and the return value of the propagation function is computed by applying the `()`-operator to the returned value of `s`.

8.6 Discussion

All domain solvers for Mozart are implemented with the CPI. The CPI provides abstractions which free the programmer from issues depending on particularities of the host system Mozart. These particularities are for example local and global propagator parameters. The programmer can entirely focus on domain solver-related issues as filter algorithms and the like. Particularly the separation of filters from to remaining solver implementation increases the degree of potential reuse of solver components significantly. The CPI is an C++-interface in a similar fashion as ILOG SOLVER but in contrast to ILOG SOLVER [73, 117, 116], it makes it possible to implement complete domain solver and not only new propagators for existing ones.

The CPI provides additional abstractions for implementing built-in domain solver as finite domains constraints [150] and finite integer set constraints (Part II) by predefined classes for access variables and domain representations. These predefined classes avoid some overhead imposed by the generality of the other interface abstractions. Section 10.3 discusses the impact of the interfaces by comparing the built-in finite domain solver with the plug-in solver presented in this chapter.

Chapter 9

Aliasing of Constraint Variables

Mozart represents equality of values directly in the constraint store. The operation equating constraint variables is called *aliasing*. Hence, taking equality of constraint variables into account requires to extend propagation engines to be able to alias constraint variables and to cope with aliased variables.

This chapter discusses the integration of aliasing of constraint variables in propagation services including the domain solver interface CPI and the impact on domain solvers. The integration is an orthogonal extension to the propagation engines presented in the previous chapters and is designed to not cause any performance penalty if not used.

The integration touches all parts of the design and the implementation of propagation engines. First the architecture of the propagation services including the domain solver interface is extended (Section 9.1). Then aliasing is integrated in propagation services (Section 9.2). Next, the way how propagation functions can benefit from aliased parameters is illustrated (Section 9.3). Finally, the implementation aspects for propagation services and for the CPI are explained (Section 9.4).

9.1 Extending the Architecture of Propagation Services

Aliasing requires to extend the propagation services architecture (Chapter 4) and the domain solver interface architecture (Chapter 6).

Propagation Services According to Section 4.2.1, a constraint variable is represented by a head and a body where the head refers to the body. Aliasing redirects the head of the *bound variable* to the head of the *remaining variable*. The head of the bound variable is transparent and both variables have the same identity. That means in the extended architecture a variable head can refer to another variable head. Thus, aliased variables are explicitly represented in the constraint store. The treatment of the domains and the connected propagators sets is up to the respective domain solver (see below).

Figure 9.1 shows aliasing of constraint variables. Variable v_1 is the bound variable and variable v_2 is the remaining variable. The domain of the remaining variable is the intersection of the domains of the aliased variables. Figure 9.1 shows only one set of

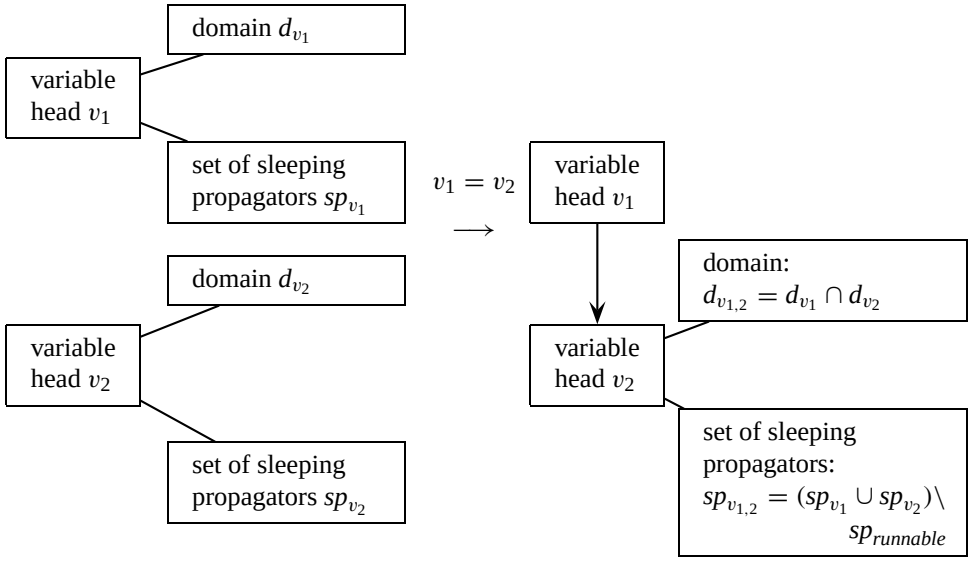


Figure 9.1: Aliasing of constraint variables.

connected propagators, but multiple sets are treated accordingly. The set of connected propagators of the remaining variable is the union of the sets of the aliased variable without the propagators scheduled ($sp_{runnable}$ in the figure) because of events caused by intersecting the domains.

Domain Solver Interface Aliasing requires a domain-dependent intersection operation which has to be provided by the domain solver. As consequence, the domain solver interface is extended to request the intersection operation from the domain solver as shown in Figure 9.2.

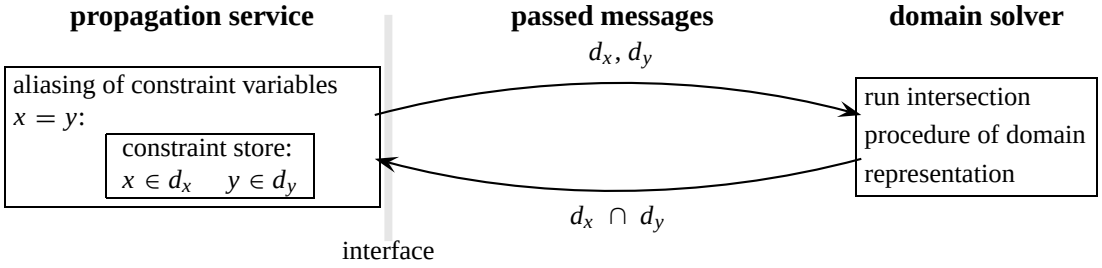


Figure 9.2: Aliasing of constraint variables.

The intersection routine is called by the aliasing routine of the propagation services and passes the involved domains (d_x and d_y) to the intersection routine and obtains the intersection of the domains ($d_x \cap d_y$).

9.2 Integrating Aliasing in Propagation Services

The integration of aliasing in propagation service has to be compatible with computation spaces and synchronization (in particular propagators). The constraint store introduced in Section 5.1.1 needs not to be extended to represent aliased variables.

The compatibility with spaces requires to trail bindings and to maintain a binding order which is compliant with stability checking of computation spaces. The binding order (see [128, Section 13.6] and [97, Section 7.3.4]) ensures that a less global variable is bound to a more global variable.[1] This subsumes that a global variable stays global and is not replaced by a local one.[2]

Threads and propagators can synchronize on constraint variables to become aliased. The suspensions for the "aliased event" are stored in the suspension list of a constraint variable which it inherits from a variable (see Section 5.2.1). Persistent propagator suspensions need extra care since they are not removed when the corresponding propagator is scheduled. Propagator suspensions are copied *resp.* moved to the remaining variable to keep the synchronization imposed on the variables.

Aliasing constraint variables proceeds in three steps:

1. Compute the intersection of the domains of the variables. Signal failure if the computed domain is empty.
2. Bind variables according to their situatedness:

   local *vs.* local Bind one variable to the other one. An optimization imposes a total order on all variables and binds always in a certain order to avoid reference chains. Move all suspension and event sets to the remaining variable.

   local *vs.* global Bind local variable to global one. Move all suspension and event set to global variable.

   global *vs.* global Bind the more local variable to the more global variable. In case both variables have the same space and the computed domain is equal to one of the domains of the global variables, bind toward the variable with the equal domain to avoid extra trailing (see condition of trailing at step 3). Otherwise the optimization mentioned for the local-local is applied. Copy all suspension and event sets to the remaining variable.

   Binding a global constraint variable has to be trailed as for a conventional global variable.
3. Update domain of the remaining constraint variable. If the remaining variable is global, trail the update if the computed domain is not equal to the domain of the remaining variable. Schedule propagators waiting for caused events and schedule all propagators in the suspension set inherited from the synchronized variable.

[1] A variable x is more global than a variable y if x's space is closer to the root space than y's space. This condition ensures that the space of the less global variable is not detected as stable too early (see [128, Section 13.6.1] for a detailed discussion).

[2] The aliasing procedure given by Würtz in [150, Section 11.3] does it not correctly since it introduces local variables.

9.3 Aliased Parameters in Propagation Functions

Aliased variables occur as aliased parameters in propagation functions of propagators. The implementation of access variables (see Section 9.4) ensures that access variables referring to aliased parameters behave identically.

This section discusses extensions of the CPI to take aliased parameters into account.

Extending Access Variables Access variables have an additional ==-operator to check if two access variables are connected to the same constraint variable in the store. Further, the CPI provides a class `OZ_CtVarVector` for efficiently (in linear time) checking for aliased parameters. Function

```
int * OZ_CtVarVector::find_equals(int * pa)
```

updates (and returns) the integer vector `pa` with the indexes (starting with 0) of the first occurrences of individual constraint variables in the vector. A non-variable is indicated by -1. As example, consider the vector $x = \langle a, b, 2, a\rangle$ with the variables a and b and the integer value (singleton) 2. Then `pa` is updated to $\langle 0, 1, -1, 0\rangle$.

Detecting Aliased Parameters Detecting aliased parameters can be beneficial for detecting failure earlier. Regard the *alldiff*-constraint $alldiff(x_1, \ldots, x_n) \equiv \forall i, j \in \{1, \ldots, n\} : i \neq j \wedge x_i \neq x_j$ which fails if two variables x_i and x_j are aliased even if x_i and x_j are not yet determined: $alldiff(x_1, \ldots, x_n) \wedge \exists i, j : x_i = x_j \longrightarrow \bot$.

Using `find_equals()` makes the implementation of the simplification rule of the *alldiff*-constraint straightforward:

```
int * e = x.find_equals();
for (int i = n; i-; )
  if ((e[i] != i && e[i] >= 0)) {
    fail propagator
  }
```

As soon as e_i is not i and not less than 0, then the access variable x_i is a multiple occurrence of the variable x_{e_i} in the store and the **`if`**-statement fires and fails the propagator.

9.4 Implementation Aspects

The implementation of propagation services is extended for aliased constraint variables in three ways (i) the aliasing procedure, (ii) detection of aliased parameters by access variables, and (iii) detection of aliased parameters in vectors of access variables.

9.4.1 Aliasing Procedure for Constraint Variables

The aliasing procedure for constraint variables is hooked into the unification procedure of Mozart which is used to impose equality on sub-graphs of the value graph in the constraint store (see Section 5.1.1) by adding necessary edges to the value graph using binding operations.[3] The aliasing procedure implements the algorithm presented in

[3] Unification is explained in detail in Mehl's thesis [88, Section 2.6.1].

Section 9.2.

9.4.2 Handling of Aliased Parameters by Access Variables

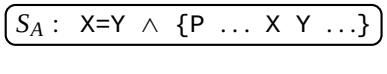
The implementation of the CPI is mainly independent from the underlying virtual machine apart from access variables. Access variables have to be able to handle aliased parameters which can be local, global and encapsulated variables. All these cases are reflected in the implementation scheme for access variables presented in this section.

The CPI has to ensure that various parameters accessing the same domain variable compute on the the same domain representation. This is straightforward for local parameters but not for global and encapsulated parameters. Furthermore, dropping parameters requires to be informed about how many references to a single variable exist. To detect this in linear time, a tagging scheme is used that adds to `OZ_CtVars` a pointer to every domain variable `CtVariable` (see Section 5.2.1). This pointer is accommodated in the field `cpi`. The implementation shares the field `cpi` with an already existing field which is saved as soon as an instance of `OZ_CtVar` and the like accesses a domain variable and restored when `leave()` or `fail()` is applied (see Section 8.5.1). That a domain variable is accessed by the CPI is memorized in a spare single bit of the domain variable.

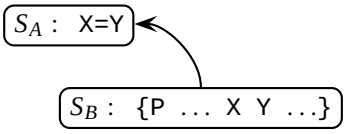
Additionally to the data structure presented in Figure 8.2 on page 74, there are three extra fields: field `var` to the domain variable, `forward` points to the access-variable which accessed the domain variable first (called *first access variable*), and `nbref` denotes the number of references to the domain variable. Note that only the first access-variable is fully initialized and its `forward`-field refers to itself.

In the following, three different situations of how domain variables can be accessed are discussed. Common to all situations is that the `cpi`-field of the domain variable and the `forward`-fields of the access-variables point to access variable `X`. Furthermore, `nbref` of `X` is set to 2 and `nbref` of `Y` is not incremented.

Local Parameters Accessing parameters referring to local variables is straightforward. Suppose aliased variables `X` and `Y` are parameters of propagator `P` in a space S_A (see figure on the right). Since both variables are local the field `domain` points directly to the domain representation in the store, as shown in Figure 9.3.



Global Parameters The initialization of an access-variable by functions `read()` or `readEncap()` for a global domain variable (see figure on the right) copies the constraint d to the field `copy` of the access-variable and refers `domain` to the copy. This means that domain reduction is performed on this copy of d and when calling `leave()` and d was transformed to d' by removing elements, d' is told to the domain variable using function `imposeConstraint()` (Program 7.6). Initialization of `Y` directs `domain` to the field `copy` of `X` (Figure 9.4) to ensure that domain reductions on parameters referring to the same domain variable are performed on the same constraint representation.



Global and Encapsulated Parameters The last situation deals with two aliased global

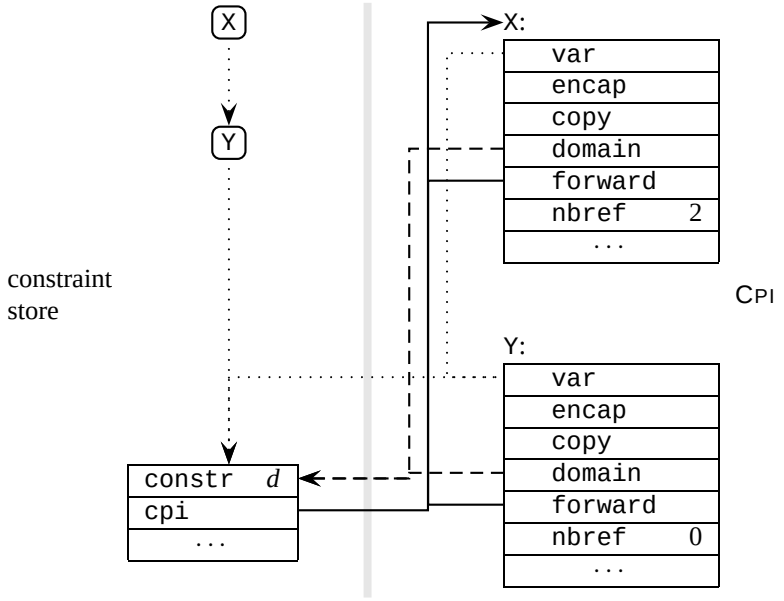


Figure 9.3: Accessing aliased local variables by the CPI .

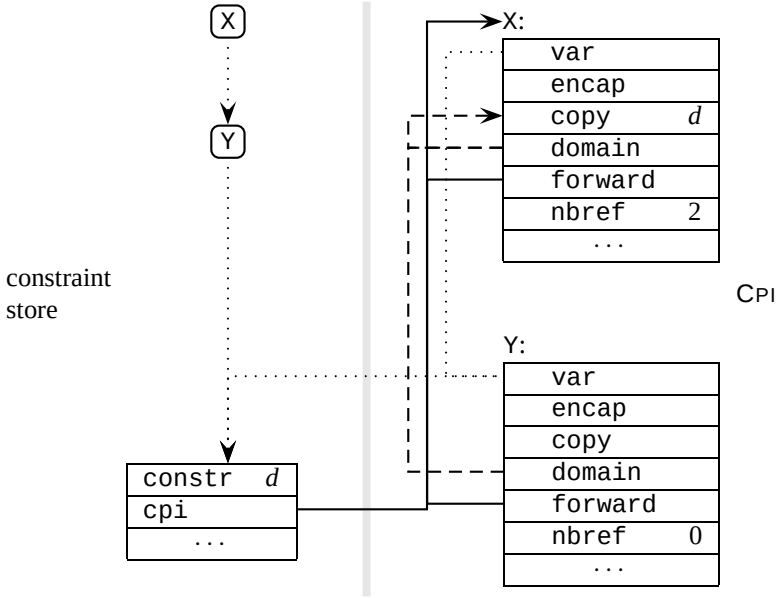
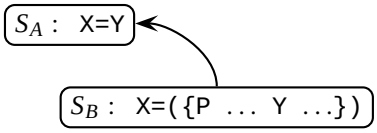


Figure 9.4: Accessing aliased global variables by the CPI .



variables where one of them (X) is determined for encapsulated propagation as it occurs for reified constraints (see figure on the right). This example clarifies why there are two distinct fields for dealing with global variables (copy) and encapsulation (encap). The implementation of the CPI maintains the

invariant to have one access-variable to represent all parameters referring to the same variable. Since a single variable in the store can occur as a global parameter and an encapsulated parameter, the field `copy` is already used for the global parameter and the field `encap` is needed. Note that this does not increase the memory consumption since access-variables are allocated in a kind of "short-term memory" only active while the propagation function is running.

Upon initialization of encapsulated access-variable `X`, the constraint *d* of the domain variable is copied to field `encap` of access-variable `X` and `domain` refers to `encap` (see Figure 9.5). Initialization of an encapsulated parameter with a local variable is the same.

The initialization of `Y` copies the domain variable's *d* to the first access-variable, *i.e.* `X` and refers field `domain` of `Y` to the copy `copy` in `X`.

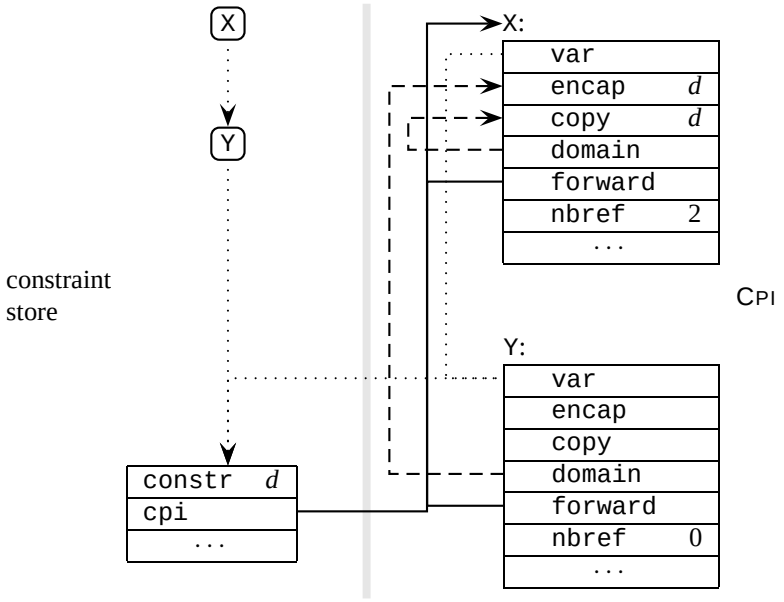


Figure 9.5: Accessing aliased global and encapsulated variables by the CPI .

The constraints of the encapsulated parameter `X` are not told to the store whereas the constraints of `Y` are told by function `imposeConstraint()` (Program 7.6).

Limitations The presented implementation of access variables is limited in the sense that instances of `OZ_CtVar` are not allowed to be assigned to each other. This may be a undesired since filters may want to partition sets of parameters. This situation can be cured by introducing an extra level of indirection: instances of class `OZ_CtVar` are accessed by pointers to them. These pointers can be hidden by class `OZ_CtVarVector` and vectors of access variables can be easily partitioned by pointer assignement.

9.4.3 Detection of Aliased Parameters in Vectors

The implementation of the function `OZ_CtVarVector::find_equals()` uses the `forward`-field of an access variable to compute the corresponding index in the returned

index vector. Access variables in a vector are initialized starting from index 0. Since the `forward`-field of an access variable points to the first access variable of a parameter, the order of initialization maintains the invariant that every `forward`-field points to an access variable in the vector with a smaller or equal index. Due to this invariant, the index of the first occurrence in the vector is simply the difference between the address of the first access variable (*i.e.*, the `forward`-field) and the address of the first access variable in the vector.

9.5 Discussion

This chapter shows that aliasing constraint variable is orthogonal to propagation services and this orthogonality is reflected in the architecture and the implementation. The treatment of aliased constraint variable certainly imposes some computational overhead, but this chapter proposes implementation techniques to minimize this overhead. That this was successful can be seen by the benchmark results in Section 10.2.

Chapter 10

Comparison and Evaluation

This chapter consists of three parts. The first part compares Mozart Oz qualitatively to other solvers while the second part compares the propagation performance of Mozart Oz with other solvers. The third part analyses the impact of the interfaces on the performance of Mozart Oz's solver.

10.1 Comparison with Other Solvers

This section compares Mozart Oz qualitatively with other solvers. Various constraint solver are characterized and relevant features are compared.

It is impossible to consider all existing constraint solvers; there are too many. Hence, solvers are considered that: (i) are easily accessible and their implementation has been presented to the scientific community, (ii) are still maintained and work out-of-the-box, and (iii) present the state-of-the-art. Finally, to be considered, a solver must be able to run the benchmarks in Section 10.2 with reasonable effort.

Other Solvers

Solvers are provided either as part of a programming language (typically Prolog) or as a library for a certain host language (typically C++).

Programming Languages The constraint solvers of the following Prolog systems are considered: GNU PROLOG 1.2.1 [38] (which is the successor of `clp(FD)` [32]), SICSTUS 3.8.5 [74], and *ECL^{i}PS^e* 5.2 [76]. Other Prolog-influenced solvers are `cc(FD)` [145] and CHIP [40].

The closest relative to Mozart is AKL(FD) [22, 20] where the constraint solver is integrated in a committed-choice language too. AKL(FD) implements an indexical-based finite domain solver in a concurrent constraint setting with encapsulated computation. Hence, there are similarities in the handling of variables and suspensions of different computation spaces. The integration of constraints is not as tight as in Oz. But unfortunately, AKL(FD) is no longer maintained.

Using Prolog as host language determines the available search strategies to depth-first search. An in-depth discussion of the search facilities of various systems can be found in [128].

Libraries Constraint libraries provide abstraction to implement constraint solver, *i.e.*, search engines and propagation engines. The first constraint libraries, as *e.g.* SCREAMER [136, 135], were implemented in Lisp (including ILOG SOLVER's predecessor PECOS [114]). Nowadays, the dominating host language is C++. The commercial C++-library ILOG-solver [73, 117, 116] is considered which can be used as target for the high-level language OPL [142] making it possible to express problems in terms of set expressions and predefined search strategies.

There are constraint libraries developed as research platforms. The C++ constraint library FIGARO [64] is designed to experiment with various search schemes and constraint solving techniques. The finite domain library CHOCO [83] is developed as platform of experimentation and targeted to the language CLAIRE [26]. CLAIRE is successfully used to solve hard combinatorial problems [29, 28].

Comparison of Features

Constraint Domains All mentioned constraint solvers support finite domain constraints. Table 10.1 given an overview over other domain solver available.

domain solver	**Mozart**	*ECL^{i}PS^e*	**SICSTUS**	**GNU PROLOG**	**ILOG**
finite domain	✓	✓	✓	✓	✓
finite Herbrand sets		✓			
finite integer sets	✓	✓			✓
rationals		✓	✓		
reals	✓	✓	✓		✓

Table 10.1: Overview of available domain solvers.

Gervet pioneered finite set constraints by introducing finite sets over Herbrand terms [55]. She implemented the solver CONJUNTO for *ECL^{i}PS^e* which in the meantime is replaced by the finite integer set solver `fd_set` [75]. See for further discussion Section 13.4.

ECL^{i}PS^e and SICSTUS provide additionally for an implementation of constraint handling rules (CHR) ([51], see Section 14.5 for a brief discussion).

Mozart provides constraints over reals as plug-in library [104] and additionally record constraints [123, 138].

Constraints Constraints with sophisticated filter algorithms are the key for solving hard combinatorial problems. They were pioneered by CHIP [10, 11]. All considered constraint solvers provide propagators with sophisticated filter, especially for scheduling applications, except GNU PROLOG. ILOG provides an extra supplementing library, called ILOG SCHEDULER [72].

Additionally, all considered solvers provide propagators for reified constraints.

Hybrid Solver A solver consisting of cooperating sub-solvers is called a *hybrid solver* [13, 121]. Typically, mathematical programming solvers (as CPLEX [70]) are connected to propagation-based constraint solvers. ILOG provides through its CONCERT-technology [71] a uniform C++-interface to propagation-based constraint solvers (ILOG SOLVER) and mathematical solvers (CPLEX).

ECL^{i}PS^e provides an interfaces to CPLEX by its library EPLEX [75]. Further, it provides the concept of a so-called *simplex daemon* which collects linear constraints and re-solves these constraint whenever their bounds change or new constraints appear [122].

Mozart is able to cooperate with other solvers. It encapsulates a solver in a propagator. The cooperation with a CPLEX-solver is presented in [104, Chapter 3].

Extensibility Constraint solvers have to be extensible to meet the requirements of increasingly demanding applications. Table 10.2 provides an overview of possible extensions. GNU PROLOG does not provide any these extensions and hence, is omitted.

extension	**Mozart**	*ECL^{i}PS^e*	**SICSTUS**	**ILOG**
new constraint domain	✓	✓	✓	
new (global) constraints	✓	✓	✓	✓
language	C++	PROLOG	PROLOG	C++

Table 10.2: Overview of possible extensions.

ILOG makes it only possible to implement new constraints for existing constraint domains. New constraints are defined by C++-classes which naturally implement global constraints. Additionally, there is dedicated support for implementing reified constraints[1].

Apart from the CPI, Mozart provides built-ins for watching finite domains. Such built-ins suspend the computation of a thread until a certain event occurred on a finite domain variable. This makes it possible to implement finite domain propagators by recursive procedures in Oz itself. This can be useful for implementing prototypical propagators. The possibility that a propagator replaces itself by another one makes the implementation of propagators for reified constraints straightforward. As for ILOG, representing a propagator by an instance of a C++-class supports naturally the implementation of constraints with sophisticated filters.

The library FIGARO supports the same extensions as ILOG but implements propagator in same fashion as Mozart. Additionally, it features a filter interface called GIFT [111] which implements the same concepts as proposed in Section 8.5.2. That makes the reuse of filters straightforward.

ECL^{i}PS^e and SICSTUS provide an attributed variable interface [67] for implementing new constraint domains. The programmer has to take care of handling sleeping constraints and the like. In contrast, the CPI fulfills this task in a self-acting way. Ad-

[1] The ILOG terminology for a *reified constraint* is *meta constraint*.

ditionally, ECL^iPS^e provides a library called RANGE [75] which simplifies the implementation of numerical domain solvers. SICSTUS provides an additional interface for implementing finite domain propagators [24].

ILOG and Mozart implement constraints by C++-propagator [117, 116, 108] in contrast to ECL^iPS^e, SICSTUS, and GNU PROLOG which use indexicals [145].

10.2 Benchmarking Propagation Efficiency

The conducted benchmarks compare the pure propagation performance of state-of-the-art constraint solvers with Mozart Oz 1.2.0. The propagation performance is measured by imposing an inconsistent constraint and measuring the time until the inconsistency is detected.

10.2.1 An Inconsistent Benchmarking Constraint

This evaluation simulates typical situations of solvers by varying the number of imposed propagators and the number of runnable propagators. Accordingly, the used benchmarking constraint can be configured to impose a given number of propagators and to have a given number of propagators runnable at a time.

The constraint has to detect the inconsistency in a predictable number of propagator invocations. Hence, the constraint is over a domain of discrete values, namely a finite domain constraint. The time to set up the solver is neglected since it is small against the time to detect the inconsistency. Further, it is possible to implement the constraint with reasonable effort for various solvers and the used propagators have to be primitive in the sense that no sophisticated propagation algorithms are used. The idea for the constraint is a closed chain of $<$-constraints. To control the number of runnable propagators, several closed chains are connected via a maximum-constraint such that to detect the inconsistency propagation of all chains is required:

$$\mathit{inconsistent}(m,n) : z = \mathit{less}(n, x_1, z) \wedge \ldots \wedge z = \mathit{less}(n, x_m, z) \wedge \mathit{maxN}(z, x_1, \ldots, x_m) \tag{10.1}$$

$$\mathit{less}(n, x_1, y) : x_{n+1} < y \ \wedge \ i \in \{1, \ldots, n\} : \ x_i < x_{i+1} \tag{10.2}$$

$$\mathit{maxN}(y, x_1, \ldots, x_n) : y = \max(x_1, y_1) \wedge \ldots \wedge y_{n-1} = \max(x_{n-1}, y_n) \wedge y_n = x_n \tag{10.3}$$

The constraint (10.1) consists of m chains of $<$-constraints of length n (constraint (10.2)) connected by the shared variable z and the maximum-constraint (10.3) which is realized by ternary maximum-constraints available in all regarded solvers. The time taken to detect the inconsistency depends on m and n and additionally on the initial domains of variables. The initial domains are intervals without holes having a size greater than n to make many propagator invocations necessary to detect the inconsistency.

The way the constraint works can be best understood regarding the analyzing corresponding constraint graph in Figure 10.1.

The upper bound of z is propagated starting from $x_{i,n}$ through the $<$-chains to $x_{i,1}$ where $i = 1, \ldots, m$. The maximum-constraints pass the upper bounds of the $x_{i,1}$ on to

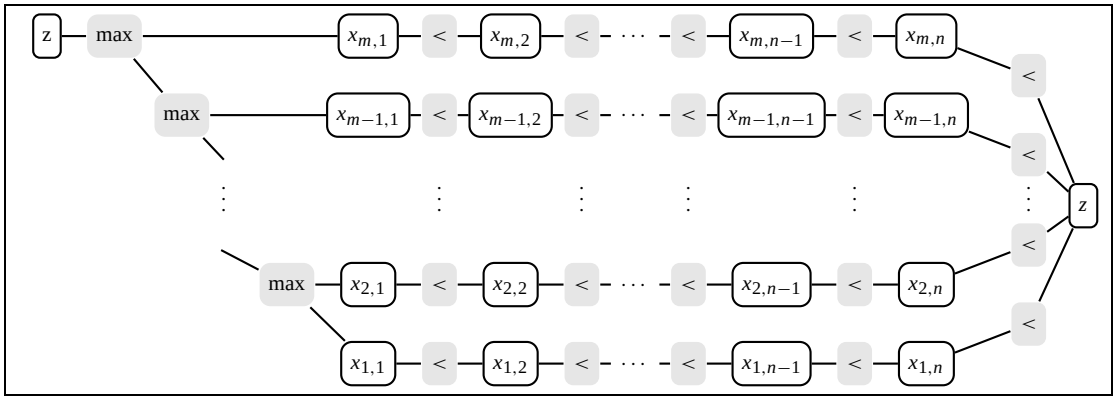


Figure 10.1: Constraint graph of the inconsistent constraint for measuring propagation performance.

z thus closing the chain. The maximum constraints pass on the upper bound only if all chains propagated through.

The number of propagators of the benchmark constraint is $m \times n + m - 1$ while the number of propagators being runnable at once is m. In the following, the number of propagators is referred to by $m \times n$ (omitting the number of maximum propagators).

10.2.2 Conducting the Benchmarks

The following systems are chosen for comparison: ILOG SOLVER 5.0, GNU PROLOG 1.2.1, SICSTUS 3.8.5, and *ECL^{i}PS^e* 5.2. These systems are used in real-world application programs and mark the state-of-the-art. The benchmarks were conducted on a machine running Linux with kernel 2.2.16-22, 256MB, AMD Athlon 700MHz.

Every benchmark was run 20 times and the arithmetic mean was used to compute the speed-ups against Mozart Oz 1.2.0. A speed-up factor greater 1 means Mozart Oz 1.2.0 is faster by that factor. Otherwise, Mozart Oz 1.2.0 is slower and the factor is given as a power of -1, *e.g.*, 2^{-1} if Mozart Oz 1.2.0 is two times slower. The results of all individual benchmarks including the corresponding variation coefficients can be found in Table A.1 on page 185 while in this chapter the results are presented in a condensed way by diagrams.

The individual benchmarks were done for all combinations of m and n where $m \times n$ does not exceed 100.000 and m and n are powers of 10. Occasionally, some solver failed for certain combinations of m and n. This is pointed out in the discussion of the individual solver comparisons.

Two types of diagrams are used: the first one depicts speed-ups *vs.* the number of propagators ($m \times n$-diagram) while the second one depicts speed-ups *vs.* the number of propagators runnable at a time (m-diagram). A vertical bar in a $m \times n$-diagram represents the arithmetic means of the speed-ups of the benchmarks for the corresponding value $m \times n$. Since a single bar typically represents the speed-ups of all combinations of m and n for a given value $m \times n$, the minimum and maximum speed-ups of these combinations are

provided for convenience. The varying speed-ups are due to the different set ups of the solvers by the parameters m and n of the benchmark constraint (10.1). The coefficients of variation are provided for the individual benchmarks in Table A.1. An m-diagram is analog to a $m \times n$-diagram. A vertical bar in a m-diagram represents the arithmetic means of all speed-ups for the corresponding value m while the minimum and maximum speed-ups for varying values of n for a given m are provided analogously to $m \times n$-diagrams.

The programs implementing the benchmark constraint for the regarded solvers can be found at [105].

Benchmarking against ILOG SOLVER ILOG SOLVER 5.0 is the only solver (apart from Mozart Oz) that is able to handle all benchmarks. But for $m \times n = 1 \times 100.000$ and $m \times n = 10 \times 10.000$, ILOG SOLVER shows a significant performance loss of being 244 *resp.* 26 times slower than Mozart Oz 1.2.0 (see right-most column in Table A.1 on page 185). These benchmarks are excluded from the diagrams.

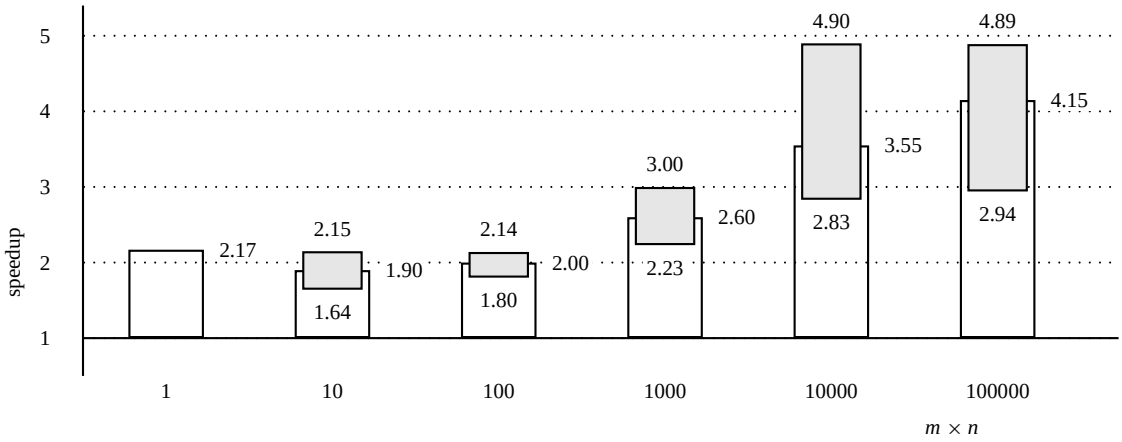


Figure 10.2: Benchmark results of ILOG SOLVER 5.0 ($m \times n$-diagram).

Figure 10.2 shows that ILOG SOLVER 5.0 performs worse as the number of propagators increases. Otherwise, Figure 10.3 shows that the number of propagators being runnable at a time does not have a direct impact on the solver's performance. To summarize, ILOG SOLVER 5.0 is in general about 2 to 4 times slower than Mozart Oz 1.2.0.

Benchmarking against GNU PROLOG GNU PROLOG 1.2.1 performs very well on the conducted benchmarks and is in one third of the cases between 2.3 up to 3.7 times faster than Mozart Oz 1.2.0. In another third GNU PROLOG 1.2.1 and Mozart Oz 1.2.0 are equally efficient, *i.e.*, speed-ups are between 1.6 and 1.5^{-1}.

GNU PROLOG 1.2.1 does not show a performance loss if the number of propagators runnable at a time increases (Figure 10.5). Unfortunately, GNU PROLOG 1.2.1 is not able to handle the benchmarks where $m \times n = 100.000$ and crashes. Hence, those tests have been ignored in the charts in Figure 10.4 and Figure 10.5. GNU PROLOG 1.2.1 provides one of the most efficient finite domain solvers but fails to run the benchmarks

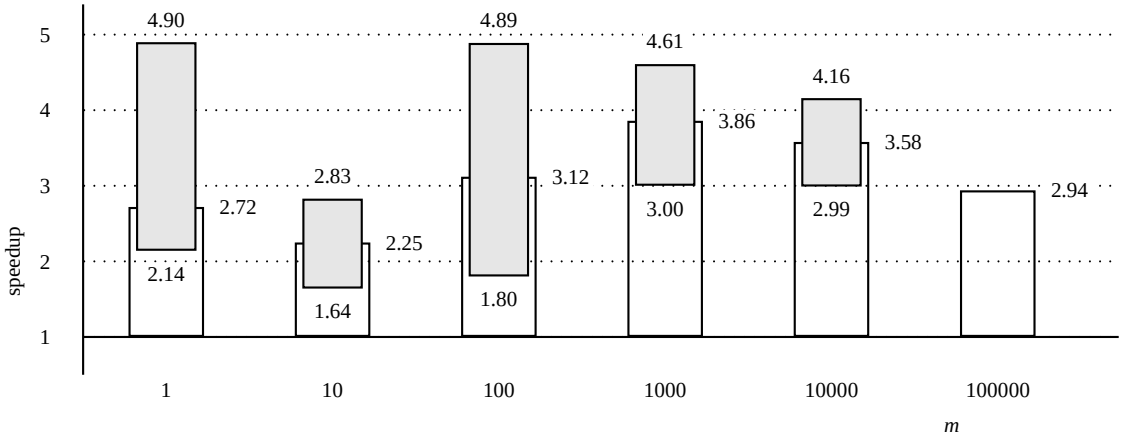


Figure 10.3: Benchmark results of ILOG SOLVER 5.0 (m-diagram).

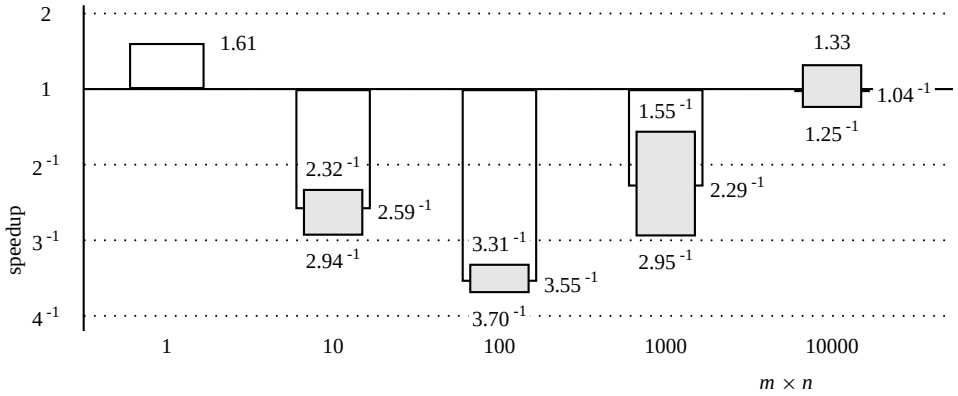


Figure 10.4: Benchmark results of GNU PROLOG 1.2.1 ($m \times n$-diagram).

with 100.000 propagators.

Benchmarking against SICSTUS PROLOG The benchmark results for varying m showed such a great deviation that they could not be used to judge the solver. Hence, m was fixed to 1 to obtain meaningful results. Furthermore, if n was greater than the size of the initial domains of the variables, it took significantly longer than the time to impose the propagators to detect the inconsistency. This is peculiar since according to personal communication with Carlsson [23] every propagator performs an initial run of the propagation function which ought to detect the inconsistency immediately in this case. Figure 10.6 shows the results for $m = 1$.

The solver produced a memory fault for $n = 100.000$. The variation of the speed-ups is significant which suggests unpredictable solver behavior. For $n = 10, 100, 1000$ the performance is comparable, *e.g.*, to ILOG SOLVER 5.0.

Benchmarking against ECL^iPS^e ECL^iPS^e 5.2 shows the same behavior as SICSTUS

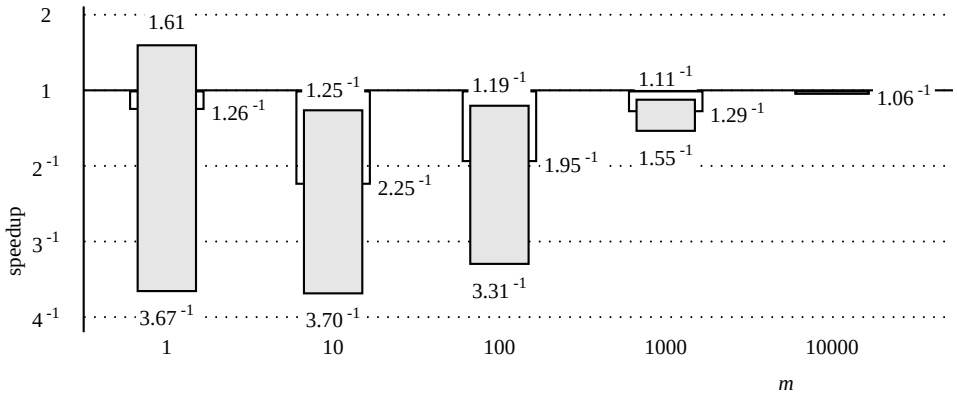


Figure 10.5: Benchmark results of GNU PROLOG 1.2.1 (m-diagram).

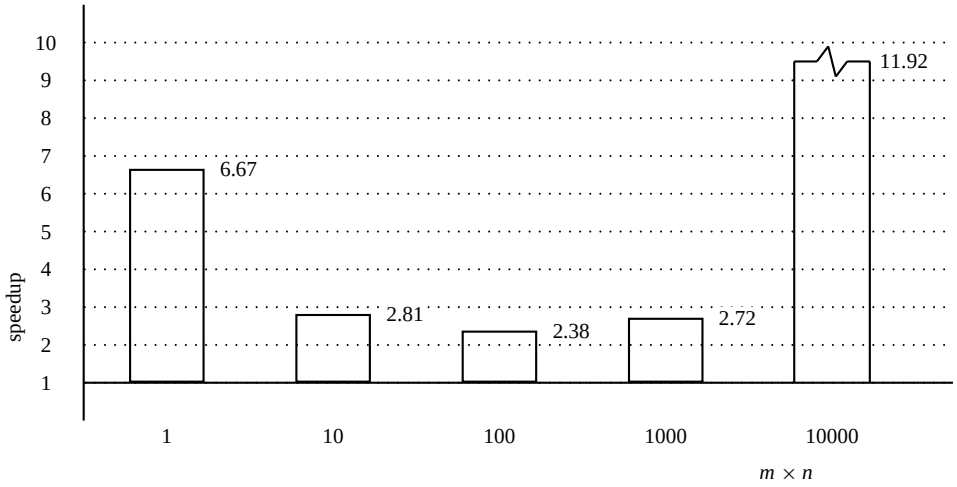


Figure 10.6: Benchmark results of SICSTUS 3.8.5 ($m \times n$-diagram).

for varying m so that m was fixed to 1 to obtain stable results. For $n = 100.000$ the solver failed to detect the inconsistency.

The results in Figure 10.7 show that the propagation performance of Mozart Oz 1.2.0 is at least 8 times faster than ECL^iPS^e 5.2. For an increasing number of propagators, the ratio is getting worse for ECL^iPS^e.

Discussion The implementations of ILOG SOLVER and Mozart are quite similar. Both solvers are written in C++ and use C++ objects for propagators to represent non-basic constraints. One reason for the better propagation performance of Mozart might be that state restoration in Mozart is completely orthogonal to the constraint propagation and thus, does not impose any computational overhead. The implementation of state restoration of ILOG might impose an overhead which worsens the plain propagation performance.

GNU PROLOG is implemented in C and uses indexicals to represent non-basic con-

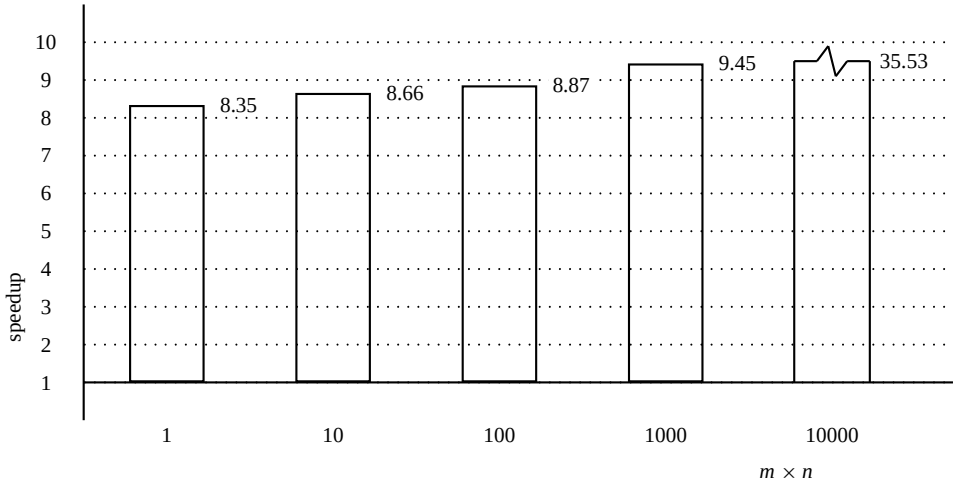


Figure 10.7: Benchmark results of ECL^iPS^e 5.2 ($m \times n$-diagram).

straints. This means that a single propagator is implemented by a set of independent indexicals. When a variable is constrained, only the directly concerned indexicals is re-executed and not whole propagators (doing much more computation). This can explain the better plain propagation performance of GNU PROLOG for the used benchmark constraint.

The worse plain propagation performance of SICSTUS PROLOG and ECL^iPS^e might be due to the implementation of their propagation engines in Prolog [23, 125]. The implementation of propagation engines uses heavily destructive variable updates which is in Prolog not as efficient available as in C/C++. In case of ECL^iPS^e, there is a complex scheduling engine used [125] which might not be able to demonstrate its benefits for the use benchmarks.

10.3 Computational Costs of Interfaces

This section analyzes the computational cost imposed by constraint programming interfaces. Since an interface adds an extra level of abstraction, a loss in efficiency is caused. The propagation performance of four variations of a finite domain solver implemented by the CPI is measured.

Standard solver This is standard finite domain solver of Mozart Oz having the finite domain constraint variables integrated in the virtual machine and connecting propagators by the CPI. The propagators do not use the filter interface provided by CPI-class `OZ_Filter` (Section 8.5.2).

Standard solver with separate filters This solver is identical to the Standard solver with the only difference that the filters are connected by the filter interface (class `OZ_Filter`).

External solver This solver is externally implemented, *i.e.*, finite domain constraint variables and propagators are implemented by the CPI. This situation occurs if a solver for a constraint domain is implemented which is not supported by the virtual machine. The propagators do not use the filter interface.

External solver with separate filters This solver is identical to the External solver with the only difference that the filters are connected by the filter interface.

The implementation of the external finite domain solver is presented as example in Chapter 8. The actual filter algorithms are of course identical.

The same set of benchmarks as for the comparison with other solvers (Section 10.2) is run to obtain the figures.

Table 10.3 shows the speed-ups obtained by comparing differently implemented finite domain solver for Mozart Oz.

Standard solver against	speed-up
Standard solver with separate filters	1.17^{-1}
External solver	2.21^{-1}
External solver with separate filters	2.38^{-1}

Table 10.3: Slow-down due to filter interface and pure CPI-implementation.

The filter algorithms being used in the inconsistent constraint require small computational effort. The figures quantifying the overhead imposed by the interfaces is hence an upper bound estimation, *i.e.*, in real-life applications where more complex propagators are employed, the overhead of the interfaces will be less relative to the overall computational effort.

The filter interface by class `OZ_Filter` imposes an slow-down from 7% (comparing the two different External solvers) to 17%. Note that the implementation of benchmarked CPI-class `OZ_Filter` is prototypical and is not yet as efficient as possible. Furthermore, for scheduling applications employing (computationally demanding) propagators, the overhead of the filter interface is barely measurable according to [111].

The implementation of a plug-in finite domain solver completely over the CPI causes a slow-down by about factor 2. This ratio will improve for the plug-in solver as soon as computationally expensive propagators are used. The CPI relies heavily on virtual functions. This disallows C++ compiler optimizations such as inlining which are essential for highly efficient code. But even solvers being slower by the factors shown in Table 10.3 are still more efficient than all the systems Mozart is compared with in Section 10.2 (except GNU PROLOG).

Part II

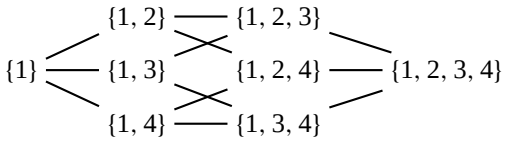
Finite Integer Set Constraints

Chapter 11

Constraint Propagation over Finite Integer Sets

This chapter discusses constraint propagation over finite integer sets. An integer set S is approximated by a lower bound s_1 and an upper bound s_2. Additional to bound constraints, sets are additionally approximated by lower and upper cardinality bounds.

The idea to approximate a set by lower and upper bounds is based on works of Gervet [55]. The lower bound denotes the elements definitely in S and the upper bound those elements possibly in S. Such a bound constraint forms a lattice shown as Hasse-diagram in the figure on the right for $s_1 = \{1\}$ and $s_2 = \{1, 2, 3, 4\}$. Additional cardinality propagation cuts off slices from the left and from the right from the lattice.



The figure makes the reason to use a bounds approximation obvious. Representing every individual element results in unfeasible exponential space demand since the power set of $s_2 \setminus s_1$ has to be represented (*i.e.*, $2^{|s_2 \setminus s_1|}$ sets have to be represented).

This chapter presents propagation rules for solving basic (Section 11.1) and non-basic (Section 11.2) finite integer set constraints and additionally, for connecting finite integer set constraints with finite domain constraints (Section 11.3). The chapter closes with an example of set constraint propagation (Section 11.4) and a discussion (Section 11.5).

11.1 Basic Constraints

Assume a set $\mathcal{U} = \{0, \ldots, \sup\}$[1] where sup is a sufficiently large integer.[2] A finite set of integers is denoted by d with $\emptyset \subseteq d \subseteq \mathcal{U}$. The complement of a set d is defined as $d^{\complement} = \mathcal{U} \setminus d$. The cardinality of a set d is denoted by $|d|$. The smallest element of a domain d is denoted with $\underline{d}$ and the largest element with $\overline{d}$. An integer is denoted by $n = 0, \ldots, \sup + 1$.

Basic finite set constraints consist of lower and upper bounds constraints in conjunc-

[1] The notation $\{a, \ldots, b\}$ denotes the set $\{e \mid a \leq e \leq b\}$.

[2] The integer constant sup is implementation-dependent and is for Mozart $2^{28} - 2$.

tion with lower and upper bounds cardinality constraints. Further, basic constraints are closed under conjunction and can be failed.

$$
\begin{array}{lll}
B ::= d_1 \subseteq S \wedge S \subseteq d_2 \quad \wedge & & \textit{bounds constraint} \\
\quad\; n_1 \leq |S| \wedge |S| \leq n_2 & & \textit{cardinality constraint} \\
\mid \; B \wedge B & & \textit{conjunction} \\
\mid \; \bot & & \textit{failure}
\end{array}
$$

Note every individual basic set constraint consists *always* of bounds and cardinality constraints and is initially $\emptyset \subseteq S \wedge S \subseteq \mathcal{U} \wedge 0 \leq |S| \wedge |S| \leq \sup + 1$. This is required to guarantee that the approximated set is finite. Furthermore, the invariants $|d_1| \leq n_1$ and $n_2 \leq |d_2|$ are maintained.

A variable S is called a *set variable* while $|S|$ is called a *cardinality variable*. A cardinality variable $|S|$ is the cardinality variable of S.

A set constraint is *determined* to d by a constraint B iff:

$$d \subseteq S \wedge S \subseteq d \in B. \tag{11.1}$$

The following derived forms are defined: $n \in S$ is $\{n\} \subseteq S$ and $n \notin S$ is $S \subseteq \mathcal{U} \setminus \{n\}$ (where $\mathcal{U} \setminus \{n\}$ is finite since $\mathcal{U}$ is finite).

Bounds Propagation Rule (11.2) detects failure if there is a basic constraint which violates the invariant that the lower bound is a subset or equal to the upper bound.

$$\frac{d_1 \subseteq S \wedge S \subseteq d_2 \wedge B}{\bot} \text{ if } d_1 \supset d_2 \tag{11.2}$$

Constraint propagation adds either elements to the lower bound (rule (11.3)) or removes elements from the upper bound (rule (11.4)).

$$\frac{d_1 \subseteq S \;\wedge\; d_2 \subseteq S \;\wedge B}{d_1 \cup d_2 \subseteq S \;\wedge B} \tag{11.3}$$

$$\frac{S \subseteq d_1 \;\wedge\; S \subseteq d_2 \;\wedge B}{S \subseteq d_1 \cap d_2 \;\wedge B} \tag{11.4}$$

Cardinality Propagation Rule (11.5) detects failure if there is a basic constraint which violates the invariant that the lower bound of the cardinality constraint less than or equal to the upper bound of the cardinality constraint.

$$\frac{n_1 \leq |S| \wedge |S| \leq n_2 \wedge B}{\bot} \text{ if } n_1 > n_2 \tag{11.5}$$

Constraint propagation either raises the lower bound of the cardinality constraint to the maximum of two individual lower bounds (rule (11.6)) or lowers the upper bound of the cardinality constraint to the minimum of two individual upper bounds (rule (11.7)).

$$\frac{n_1 \leq |S| \;\wedge\; n_2 \leq |S| \;\wedge B}{\max(n_1, n_2) \leq |S| \;\wedge B} \tag{11.6}$$

$$\frac{|S| \leq n_1 \;\wedge\; |S| \leq n_2 \;\wedge B}{|S| \leq \min(n_1, n_2) \;\wedge B} \tag{11.7}$$

Connecting Bounds and Cardinality Propagation The purpose of connecting bounds and cardinality propagation is to determine sets earlier. The following rules determine a set by combining a bound constraint and a cardinality constraint.

$$\frac{d_1 \subseteq S \wedge \boxed{S \subseteq d_2} \wedge |S| \leq n \wedge B}{d_1 \subseteq S \wedge \boxed{S \subseteq d_1} \wedge |S| \leq n \wedge B} \text{ if } |d_1| = n \wedge d_1 \subset d_2 \tag{11.8}$$

$$\frac{\boxed{d_1 \subseteq S} \wedge S \subseteq d_2 \wedge n \leq |S| \wedge B}{\boxed{d_2 \subseteq S} \wedge S \subseteq d_2 \wedge n \leq |S| \wedge B} \text{ if } |d_2| = n \wedge d_1 \subset d_2 \tag{11.9}$$

Rule (11.8) determines a set if the cardinality of the lower bound set is equal to the upper bound of the cardinality constraint. Rule (11.9) does the same for the upper bound set and the lower bound of the cardinality constraint.

$$\frac{d \subseteq S \wedge \boxed{n \leq |S|} \wedge B}{d \subseteq S \wedge \boxed{|d| \leq |S|} \wedge B} \text{ if } |d| > n \tag{11.10}$$

$$\frac{S \subseteq d \wedge \boxed{|S| \leq n} \wedge B}{S \subseteq d \wedge \boxed{|S| \leq |d|} \wedge B} \text{ if } |d| < n \tag{11.11}$$

Rules (11.10) and (11.11) maintain for a basic constraint $d_1 \subseteq S \wedge S \subseteq d_2 \wedge n_1 \leq |S| \wedge |S| \leq n_2$ the invariant $|d_1| \leq n_1 \leq n_2 \leq |d_2|$. Without this invariant, a basic set constraint C in normal form can be satisfiable *but not* failed ($\bot \notin C$) because rule (11.2) is not applicable. Note that the condition of the rules (11.10) and (11.11) are needed for termination.

11.2 Non-basic Constraints

Non-basic constraints include basic constraints, are closed under conjunction and provide the primitive non-basic constraints $S_1 \supseteq S_2 \cap S_3$ and $S_1 \subseteq S_2 \cup S_3$.

$$C \quad ::= B \mid C \wedge C \mid S_1 \supseteq S_2 \cap S_3 \mid S_1 \subseteq S_2 \cup S_3$$

All other desired non-basic set constraints are expressed in terms of C (Figure 11.1).

Bounds Propagation for $S_1 \supseteq S_2 \cap S_3$ Rule (11.12) constrains the lower bound of S_1 while rule (11.13) constrains the upper bound of S_2 *resp.* S_3.

$$S_1 \supseteq S_2 \cap S_3 : \frac{\boxed{d_1 \subseteq S_1} \wedge d_2 \subseteq S_2 \wedge d_3 \subseteq S_3 \wedge C}{\boxed{d_2 \cap d_3 \subseteq S_1} \wedge d_2 \subseteq S_2 \wedge d_3 \subseteq S_3 \wedge C} \text{ if } d_1 \subset d_2 \cap d_3 \tag{11.12}$$

Rule (11.12) states that the lower bound of S_1 contains all those elements which are in the lower bound of S_2 *and* S_3.

$$S_1 \supseteq S_2 \cap S_3 : \frac{S_1 \subseteq d_1 \wedge \boxed{d_i \subseteq S_i} \wedge d_j \subseteq S_j \wedge C}{S_1 \subseteq d_1 \wedge \boxed{S_i \subseteq (d_j \setminus d_1)^{\complement}} \wedge d_j \subseteq S_j \wedge C} \; i, j \in \{2,3\}, i \neq j \tag{11.13}$$

$$\text{if } d_i \subset (d_j \setminus d_1)^{\complement}$$

inclusion:	$S_1 \subseteq S_2$	$\equiv$	$S_1 \subseteq S_2 \cup S_3 \ \wedge \ S_3 \subseteq \emptyset$
disjoint:	$S_1 \parallel S_2$	$\equiv$	$\emptyset \supseteq S_1 \cap S_2$
union:	$S_1 = S_2 \cup S_3$	$\equiv$	$S_1 \subseteq S_2 \cup S_3 \ \wedge \ S_2 \subseteq S_1 \ \wedge \ S_3 \subseteq S_1$
complement:	$S_1 = S_2^{\complement}$	$\equiv$	$\emptyset \supseteq S_1 \cap S_2 \ \wedge \ \mathcal{U} \subseteq S_1 \cup S_2$
intersection:	$S_1 = S_2 \cap S_3$	$\equiv$	$S_1 \supseteq S_2 \cap S_3 \ \wedge \ S_1 \subseteq S_2 \ \wedge \ S_1 \subseteq S_3$
difference:	$S_1 = S_2 \setminus S_3$	$\equiv$	$S_1 = S_2 \cap S_3^{\complement}$
partition:	$S_1 = S_2 \uplus S_3$	$\equiv$	$S_1 = S_2 \cup S_3 \ \wedge \ S_1 \parallel S_2$

Figure 11.1: Expressing set constraints in terms of $S_1 \supseteq S_2 \cap S_3$ and $S_1 \subseteq S_2 \cup S_3$
.

Rule (11.13) states that the upper bound of S_i does not contain those elements which are in S_j but *not* in S_1.

Cardinality Propagation for $S_1 \supseteq S_2 \cap S_3$ Cardinality propagation is based on the number of elements m which can be distributed over S_2 and S_3 without sharing any elements. The value of m is $m = |d_2 \cup d_3|$ where $S_2 \subseteq d_2$ and $S_3 \subseteq d_3$.

The following rules propagate on the cardinality constraints:

$$S_1 \supseteq S_2 \cap S_3 : \frac{n_1 \leq |S_1| \ \wedge n_2 \leq |S_2| \wedge n_3 \leq |S_3| \wedge C}{n_2 + n_3 - m \leq |S_1| \ \wedge n_2 \leq |S_2| \wedge n_3 \leq |S_3| \wedge C} \quad \text{if } n_1 < n_2 + n_3 - m \tag{11.14}$$

The lower bound of S_1 contains at least as many elements as have to be shared by S_2 and S_3. Since m elements can be distributed to S_2 and S_3 without any sharing, the number of elements that exceeds m has to be shared (rule (11.14)).

$$S_1 \supseteq S_2 \cap S_3 : \frac{|S_1| \leq n_1 \wedge \ n_i \leq |S_i| \ \wedge n_j \leq \left|S_j\right| \wedge C}{|S_1| \leq n_1 \wedge \ |S_i| \leq m + n_1 - n_j \ \wedge n_j \leq \left|S_j\right| \wedge C} \ i, j \in \{2,3\}, i \neq j \quad \text{if } n_i > m + n_1 - n_j \tag{11.15}$$

The inference for the upper bound of the cardinality of S_i is defined by rule (11.15) which can straightforwardly derived from the right hand-side of rule (11.14) by resolving $n_1 = n_2 + n_3 - m$ to n_i.[3]

[3] An interpretation of this rule is that $ns = m - n_1$ elements are not shared between S_2 and S_3. S_2 does not share $ns_2 = n_2 - n_1$ elements with S_3 while S_3 does not share $ns_3 = n_3 - n_1$ elements with S_2. The number of elements that are only in S_2 *resp.* S_3 must exceed the maximum number of elements that are not shared: $ns \geq ns_2 + ns_3$. Expanding this in-equation yields: $m - n_1 \geq (n_2 - n_1) + (n_2 - n_1)$ which can be resolved either to $n_2 \leq m + n_1 - n_3$ or to $n_3 \leq m + n_1 - n_2$.

Bounds Propagation for $S_1 \subseteq S_2 \cup S_3$ The upper bound of S_1 is constrained by rule (11.16) to contain all those elements which are in the upper bounds of S_2 *or* S_3.

$$S_1 \subseteq S_2 \cup S_3 : \frac{S_1 \subseteq d_1 \wedge S_2 \subseteq d_2 \wedge S_3 \subseteq d_3 \wedge C}{S_1 \subseteq d_2 \cup d_3 \wedge S_2 \subseteq d_2 \wedge S_3 \subseteq d_3 \wedge C} \text{ if } d_2 \cup d_3 \subset d_1 \quad (11.16)$$

The lower bound of S_i is constrained to contain those elements e that are contained in the lower bound of S_1 but *not* S_j (see rule (11.17)). This means, that S_i has to contribute the elements e.

$$S_1 \subseteq S_2 \cup S_3 : \frac{d_1 \subseteq S_1 \wedge d_i \subseteq S_i \wedge d_j \subseteq S_j \wedge C}{d_1 \subseteq S_1 \wedge d_1 \setminus d_j \subseteq S_i \wedge d_j \subseteq S_j \wedge C} \; i, j \in \{2,3\}, i \neq j \quad (11.17)$$
$$\text{if } d_i \subset d_1 \setminus d_j$$

Cardinality Propagation for $S_1 \subseteq S_2 \cup S_3$ Cardinality propagation is performed according to the following rules:

$$S_1 \subseteq S_2 \cup S_3 : \frac{|S_1| \leq n_1 \wedge |S_2| \leq n_2 \wedge |S_3| \leq n_3 \wedge C}{|S_1| \leq n_2 + n_3 \wedge |S_2| \leq n_2 \wedge |S_3| \leq n_3 \wedge C} \text{ if } n_1 > n_2 + n_3 \quad (11.18)$$

The upper bound of the cardinality of S_1 is inferred by rule (11.18) and states that S_1 must not contain more elements than S_2 and S_3 together.

$$S_1 \subseteq S_2 \cup S_3 : \frac{|S_1| \leq n_1 \wedge n_i \leq |S_i| \wedge n_j \leq \left|S_j\right| \wedge C}{|S_1| \leq n_1 \wedge n_1 - n_j \leq |S_i| \wedge n_j \leq \left|S_j\right| \wedge C} \; i, j \in \{2,3\}, i \neq j \quad (11.19)$$
$$\text{if } n_i < n_1 - n_j$$

The propagation for the lower bound of the cardinality of S_i is that S_i has to contribute at least as many elements as lacking between S_1 and S_j, *i.e.* $n_1 - n_j$ elements (rule (11.19)).

11.3 Connecting Finite Integer Sets and Finite Domains

Finite integer sets gain extra expressiveness and propagation by combining them with finite domain constraints. This section discusses various ways of combining both constraint systems while Chapter 13 demonstrates programming techniques taking advantage of this combination.

Finite Domain Constraints Basic finite domain constraints are added by:

$$B \quad ::= \ldots \mid I \in d$$

A finite domain variable is denoted by I and denotes a single integer $n \in d$. A finite domain variable is determined to the integer n if $I \in \{n\}$.

The following rules describe propagation of basic finite domain constraints:

$$\frac{I \in d_1 \wedge I \in d_2 \wedge B}{I \in d_1 \cap d_2 \wedge B} \quad (11.20) \qquad \frac{I \in \emptyset \wedge B}{\bot} \quad (11.21)$$

Rule (11.20) combines two domains d_1 and d_2 of a single variable I while rule (11.21) detects failure.

There is a rich set of non-basic constraints available for finite domain constraints. For propagators implementing non-basic finite domain constraints refer to [46].

Cardinality The following constraint connects a finite domain variable I with the cardinality of the set S and propagates forth and back the changes to I and S.

$$C \quad ::= \ldots \mid I = |S|$$

Propagation is performed by the following rules:

$$I = |S| : \frac{n \leq |S| \wedge I \in d \wedge C}{n \leq |S| \wedge I \in d \cap \{n, \ldots, \sup\} \wedge C} \text{ if } n > \underline{d} \quad (11.22)$$

$$I = |S| : \frac{|S| \leq n \wedge I \in d \wedge C}{|S| \leq n \wedge I \in d \cap \{0, \ldots, n\} \wedge C} \text{ if } n < \overline{d} \quad (11.23)$$

Rule (11.22) propagates changes of the lower bound of the cardinality constraint of S to I while the same is done for the upper bound of S rule (11.23).

$$I = |S| : \frac{n \leq |S| \wedge I \in d \wedge C}{\underline{d} \leq |S| \wedge I \in d \wedge C} \text{ if } \underline{d} > n \quad (11.24)$$

$$I = |S| : \frac{|S| \leq n \wedge I \in d \wedge C}{|S| \leq \overline{d} \wedge I \in d \wedge C} \text{ if } \overline{d} < n \quad (11.25)$$

Rules (11.24) and (11.25) propagate changes of the cardinality constraint of S to the domain of I.

Membership As mentioned, $n \in S$ is a derived form of $d \subseteq S$. This derived form can be generalized to a non-basic constraint:

$$C \quad ::= \ldots \mid I \in S$$

Note that $I \notin S$ is not considered separately since it is already covered due to $I \notin S \equiv I \in S^{\complement}$.

Rule (11.26) propagates the upper bound of S to the domain of I while rule (11.27) propagates the determined value of I to the lower bound of S.

$$I \in S : \frac{I \in d_1 \wedge S \subseteq d_2 \wedge C}{I \in d_1 \cap d_2 \wedge S \subseteq d_2 \wedge C} \text{ if } d_1 \cap d_2 \subset d_1 \quad (11.26)$$

$$I \in S : \frac{d \subseteq S \wedge I \in \{n\} \wedge C}{d \cup \{n\} \subseteq S \wedge I \in \{n\} \wedge C} \text{ if } n \notin d \quad (11.27)$$

Reification of Membership A reified membership constraint reflects the validity of a membership constraint $I_1 \in S$ to a 0/1-variable I_2 which can be connected with finite domain constraints.

$$C \quad ::= \ldots \mid (I_1 \in S \leftrightarrow I_2) \wedge I_2 \in \{0,1\}$$

The following rules detect entailment (rule (11.28)) *resp.* failure (rule (11.29)) of $I_1 \in S$ and reflect the result to I_2:

$$\frac{I_1 \in d_1 \wedge d_2 \subseteq S \wedge \; I_2 \in \{0,1\} \; \wedge \; (I_1 \in S \leftrightarrow I_2) \; \wedge C}{I_1 \in d_1 \wedge d_2 \subseteq S \wedge \; I_2 \in \{1\} \; \wedge C} \text{ if } d_1 \subseteq d_2 \qquad (11.28)$$

$$\frac{I \in d_1 \wedge S \subseteq d_2 \wedge \; I_2 \in \{0,1\} \; \wedge \; (I_1 \in S \leftrightarrow I_2) \; \wedge C}{I \in d_1 \wedge S \subseteq d_2 \wedge \; I_2 \in \{0\} \; \wedge C} \text{ if } d_1 \cap d_2 = \emptyset \qquad (11.29)$$

Constraining the domain of I_2 to a singleton $\{1\}$ ($\{0\}$) replaces $(I_1 \in S \leftrightarrow I_2)$ by $I_1 \in S$ ($I_1 \notin S$):

$$\frac{I_2 = \{1\} \wedge \; (I_1 \in S \leftrightarrow I_2) \; \wedge C}{I_2 = \{1\} \wedge \; I_1 \in S \; \wedge C} \quad (11.30) \qquad \frac{I_2 = \{0\} \wedge \; (I_1 \in S \leftrightarrow I_2) \; \wedge C}{I_2 = \{0\} \wedge \; I_1 \notin S \; \wedge C} \quad (11.31)$$

Note that until now propagation rules only derived new basic constraints; here they replace non-basic constraints.

11.4 An Example of Set Constraint Propagation

This section explains by means of examples constraint propagation and distribution for finite integer set constraints.

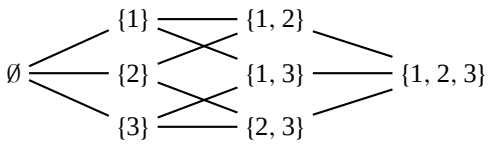
Constraint Propagation As an example for constraint propagation, assume the basic constraints

$$\emptyset \subseteq S_1 \wedge S_1 \subseteq \{1,\ldots,3\} \;\wedge\; \emptyset \subseteq S_2 \wedge S_2 \subseteq \{1,\ldots,4\}$$

in conjunction with the non-basic constraints:

$$\emptyset \supseteq S_1 \cap S_2 \;\wedge\; \{1,\ldots,4\} \subseteq S_1 \cup S_2.$$

The domain reduction is illustrated for variable S_1 by presenting the set constraint as Hasse-diagram and crossing out removed values. The initial Hasse-diagram of $S_1 : \emptyset \subseteq S_1 \wedge S_1 \subseteq \{1,\ldots,3\}$ is:

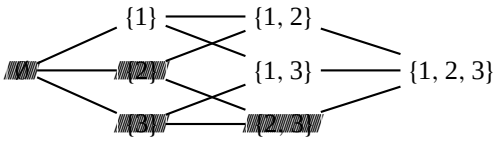


Constraint propagation of the non-basic constraints is demonstrated by showing the propagation steps taken in Table 11.1. These steps are caused by sequentially imposing the basic constraints $1 \in S_1$, $2 \notin S_2$, and $|S_1| \in \{1, 2\}$. Table 11.1 shows in column *imposed* the imposed basic constraints and in column *applied rule* the applied rules. The columns S_1 and S_2 show how the variables change as result of constraint propagation. For the sake of a compact representation, a set variable $\{1, 3\} \subseteq S \wedge S \subseteq \{1, 2, 3, 4\}$ is depicted as $\{\mathbf{1}, 2, \mathbf{3}, 4\}$.

step	imposed	applied rule	S_1	S_2
0			$\{1, 2, 3\}$	$\{1, 2, 3, 4\}$
1	$1 \in S_1$		$\{\mathbf{1}, 2, 3\}$	$\{1, 2, 3, 4\}$
2		(11.13)	$\{\mathbf{1}, 2, 3\}$	$\{2, 3, 4\}$
3	$2 \notin S_2$		$\{\mathbf{1}, 2, 3\}$	$\{3, 4\}$
4		(11.17)	$\{\mathbf{1}, \mathbf{2}, 3\}$	$\{3, 4\}$
5	$1 \leq \lvert S_1 \rvert \leq 2$	(11.8)	$\{\mathbf{1}, \mathbf{2}\}$	$\{3, 4\}$
6		(11.17)	$\{\mathbf{1}, \mathbf{2}\}$	$\{\mathbf{3}, \mathbf{4}\}$

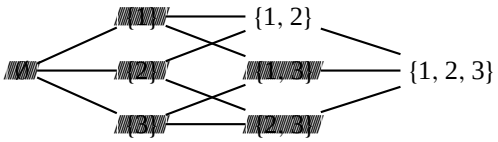
Table 11.1: Finite set constraint propagation.

Step 0 of Table 11.1 shows the initial state and step 1 adds 1 to the lower bound of S_1. The corresponding Hasse-diagram of S_1 representation is:

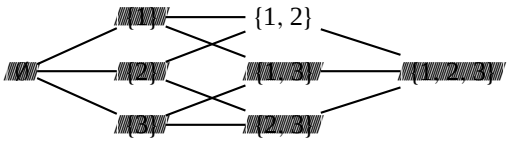


Note that all set values not containing 1 are crossed out in the Hasse-diagram, *i.e.* removed from the domain. Step 1 causes propagation rule (11.13) of constraint $\emptyset \supseteq S_1 \cap S_2$ to be applied (propagation step 2) which removes element 1 from S_2.

Step 3 drops 2 from S_2 which in turn triggers in step 4 rule (11.17) of constraint $\{1, \ldots, 4\} \subseteq S_1 \cup S_2$. This rule adds 2 to the lower bound of S_1 since it is required to form the union of $\{1, \ldots, 4\}$. The corresponding Hasse-diagram of S_1 is:



The last basic constraint is $|S_1| \in \{1, 2\}$ (step 5). It determines S_1 to $\{1, 2\}$ since it is the only value possible. This is caused by rule (11.8) for basic cardinality constraints which removes the possible value $\{1, 2, 3\}$. The Hasse-diagram makes it clear:



Finally, also S_2 is determined to $\{3, 4\}$ by rule (11.17). This rule enforces the required elements 3 and 4 to be in S_2 since they are not in S_1. In turn, this triggers rule (11.1) which determines S_2.

Distribution In general, constraint propagation is not able to infer a solution or to detect the absence of a solution. Hence, constraint propagation is supplemented with making non-deterministic distribution steps. A distribution step computes a branching constraint (Section 2.1). Alternating constraint propagation and distribution leads to tree search. Rule (11.32) describes a typical branching algorithm for a distribution step of finite integer set constraints.

$$\frac{d_1 \subseteq S \wedge S \subseteq d_2 \wedge C}{(\{n\} \cup d_1 \subseteq S \wedge S \subseteq d_2 \wedge C) \vee (d_1 \subseteq S \wedge S \subseteq d_2 \setminus \{n\} \wedge C)} \quad n \in d_2 \setminus d_1 \tag{11.32}$$

The set $d_2 \setminus d_1$ is called the *undecided set* of S. This rule picks an element n of the undecided set and creates a disjunction which includes n in S in one alternative and excludes n from S in the other alternative. The exploration algorithm determines the order of how the alternatives are explored.

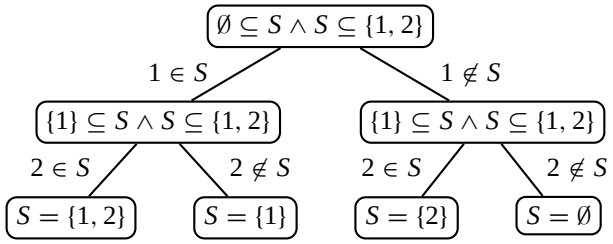


Figure 11.2: Search tree over an integer set.

Figure 11.2 shows a typical finite set search tree. The element 1 of the upper bound of S is assumed to be in (left branch) or not in (right branch). The respective subtrees continue by picking another element (2) and assuming it to be in or not in S until S is determined.

11.5 Discussion

The propagation rules presented in this chapter are normalizing, *i.e.*, if there is no rule applicable the basic constraint is in normal form. A basic constraint is in normal form if either (i) $\bot \in B$ or (ii) $B = \bigwedge_i d_i \subseteq S_i \wedge S_i \subseteq d_i' \wedge n_i \leq |S_i| \wedge |S_i| \leq n_i'$, all S_i are pairwise distinct and $\forall i : \; d_i \subseteq d_i' \wedge n_i \leq n_i'$. In case of (i), B is failed. In case $\forall i : \; d_i = d_i' \wedge n_i = n_i'$, B is solved.

Bound reasoning and cardinality reasoning are orthogonal. They can be independently implemented as long as the non-basic constraint $I = |S|$ is available. A set is then represented by two constraint variables: a bound variable and a cardinality variable.

Mozart uses a single variable with bound and cardinality information for the sake of efficiency.

All available finite set constraint solver provide different degrees of cardinality reasoning. CONJUNTO adds cardinality information to the basic constraints by lower and upper bounds while ILOG SOLVER associates a finite domain variable to every finite set variable. Thus, the value of a set can be earlier determined by rules (11.8) and (11.9). Non-basic constraints of CONJUNTO as well as CONJUNTO's successor `fd_sets` [125] do not perform cardinality reasoning. In contrast, the commercial ILOG SOLVER library [119] and the Cardinal-solver for ECL^iPS^e of Azevedo and Barahona [8] perform cardinality reasoning similar to what is described in this chapter. Azevedo and Barahona demonstrate in [8] the benefits of cardinality reasoning by benchmarks on digital circuit diagnosis which show speed-ups of more than an order of magnitude against an equivalent solver without cardinality reasoning.

Set constraints discribed in this chapter lead to (classical) CSPs covered by the semi-ring constraint solving framework presented in [17, 18, 16]. Thus, the local consistency properties of the semi-ring framework are inherited.

Chapter 12

Construction of Filter Algorithms

This chapter presents a scheme for automatically constructing filter algorithms for set constraints which are conjunctions of the primitive non-basic constraints $S_1 \supseteq S_2 \cap S_3$ and $S_1 \subseteq S_2 \cup S_3$. These two constraints make it possible to express, for example, standard set operations as shown in Figure 11.1 on page 104.

The proposed scheme constructs from a given set constraint an idempotent filter (see Section 2.2) which updates basic constraints with constraint projectors. A constraint projector modifies directly the constraints in the store and is part of the actual implementation. A constraint projector causes propagation events by modifying the store *and* is dependent on propagation events of other projectors. These mutual dependencies make the construction of filters non-trivial.

The generation of a filter proceeds in three steps.

Derive Constraint Projectors (Section 12.1) Express the constraint to be realized by a conjunction of constraints $S_1 \supseteq S_2 \cap S_3$ and $S_1 \subseteq S_2 \cup S_3$. Transform the conjunction to the corresponding set of constraint projectors.

Derive Events (Section 12.2) Compute for every constraint projector the events it causes by updating and the events it depends on.

Generate Filter (Section 12.3) Detect dependencies among the projectors by creating a dependency graph based on the events computed in the previous step and computing the strongly connected components of this graph. Collect mutually dependent projectors in loops to obtain propagation fixed-point of these projectors. Order loops and non-mutually dependent projectors such that dependencies and execution order are compatible.

This chapter closes by illustrating the generation of a filter for the constraint $S_1 = S_2 \cap S_3$ in Section 12.4 and a discussion in Section 12.5.

An implementation of the proposed scheme (called filter generator) generates filter algorithms in C++ which can be directly used by propagators implemented by the CPI of Chapter 8. The example filters presented in Section 12.4 were generated by the filter generator. The filter generator and generated C++-filters for various set constraints are available at [105].

12.1 Computation of Constraint Projectors

A constraint projector updates a component (d_1, d_2, n_1 or n_2) of a basic set constraint $d_1 \subseteq S \wedge S \subseteq d_2 \wedge n_1 \leq |S| \wedge |S| \leq n_2$.

The components d_1, d_2, n_1 and n_2 of the basic constraint can be retrieved by the following access functions: $d_1 = \lfloor S \rfloor$, $d_2 = \lceil S \rceil$, $n_1 = \underline{|S|}$ and $n_2 = \overline{|S|}$.

Basic constraints are imposed on a constraint variable by updating the components of the constraints. The following constraint projectors are defined:

constraint projector $::= \lfloor S \rfloor \mathrel{\dot{\supseteq}} E^S$	*update* $\lfloor S \rfloor$ *to* $\lfloor S \rfloor \cup E^S$
$\mid \quad \lceil S \rceil \mathrel{\dot{\subseteq}} E^S$	*update* $\lceil S \rceil$ *to* $\lceil S \rceil \cap E^S$
$\mid \quad \underline{\lvert S \rvert} \mathrel{\dot{\geq}} E^C$	*update* $\underline{\lvert S \rvert}$ *to* $\max(\underline{\lvert S \rvert}, E^C)$
$\mid \quad \overline{\lvert S \rvert} \mathrel{\dot{\leq}} E^C$	*update* $\overline{\lvert S \rvert}$ *to* $\min(\overline{\lvert S \rvert}, E^C)$

where E^S is a set expression denoting a set and is of the form:

$$E^S ::= \lfloor S \rfloor \mid \lceil S \rceil \mid d \mid \mathcal{U} \mid E^S \cup E^S \mid E^S \cap E^S \mid E^S \setminus E^S$$

and E^C is a cardinality expression denoting an integer and is of the form:

$$E^C ::= \underline{|S|} \mid \overline{|S|} \mid n \mid \sup \mid \underline{\left|E^S\right|} \mid E^C + E^C \mid E^C - E^C \mid \max(E^C, E^C) \mid \min(E^C, E^C)$$

The application of a projector may abort a filter algorithm with failure if either rule (11.2) or (11.5) fires.

The set variable S *resp.* cardinality variable $|S|$ on the left hand-side of a projector p is called *projection variable* of p. Applying a projector $\dot{\leq}$ or $\dot{\geq}$ to a set variable S means implicitly the application to the cardinality variable of S. The set of variables occurring in a set expression E^S is denoted by $\mathcal{V}(E^S)$ while the set of variables occurring in a cardinality expression is denoted by $\mathcal{V}(E^C)$. Note that $\mathcal{V}(E^S)$ contains only set variables and the set $\mathcal{V}(E^C)$ contains set and cardinality variables.

The projectors for the primitive non-basic constraints $S_1 \supseteq S_2 \cap S_3$ and $S_1 \subseteq S_2 \cup S_3$ are derived from the gray-shaded parts of the conclusions of the propagation rules (11.12)–(11.19) in Section 11.2. A set of constraint projectors is denoted by $\mathcal{P}$.

Constraint Projectors for $S_1 \supseteq S_2 \cap S_3$ Projector (12.1) updates the lower bound set of S_1 and is derived for propagation rule (11.12). Projectors (12.2) and (12.3) are derived from propagation rule (11.13) and update the upper bound sets of S_2 and S_3.

The lower bound of the cardinality constraint of S_1 is updated by projector (12.4) which is derived from propagation rule (11.14). The upper bounds of the cardinality constraints of S_2 and S_3 are updated by projectors (12.5) and (12.6) which are derived from propagation rule (11.15).

$$\lfloor S_1 \rfloor \mathrel{\dot{\supseteq}} \lfloor S_2 \rfloor \cap \lfloor S_3 \rfloor \qquad (12.1)$$

$$\lceil S_2 \rceil \mathrel{\dot{\subseteq}} (\lfloor S_3 \rfloor \setminus \lceil S_1 \rceil)^{\complement} \qquad (12.2)$$

$$\lceil S_3 \rceil \mathrel{\dot{\subseteq}} (\lfloor S_2 \rfloor \setminus \lceil S_1 \rceil)^{\complement} \qquad (12.3)$$

$$\underline{|S_1|} \mathrel{\dot{\geq}} \underline{|S_2|} + \underline{|S_3|} - |\mathcal{U}| \qquad (12.4)$$

$$\overline{|S_2|} \mathrel{\dot{\leq}} |\mathcal{U}| + \overline{|S_1|} - \underline{|S_3|} \qquad (12.5)$$

$$\overline{|S_3|} \mathrel{\dot{\leq}} |\mathcal{U}| + \overline{|S_1|} - \underline{|S_2|} \qquad (12.6)$$

The set of projectors for the constraint $S_1 \supseteq S_2 \cap S_3$ is denoted by $\mathcal{P}_{S_1 \supseteq S_2 \cap S_3}(S_1, S_2, S_3)$ where the parameters are accordingly replaced.

Constraint Projectors for $S_1 \subseteq S_2 \cup S_3$ Projector (12.7) updates the upper bound set of S_1 and is derived for propagation rule (11.16). Projectors (12.8) and (12.9) are derived from propagation rule (11.17) and update the lower bound sets of S_2 and S_3.

The upper bound of the cardinality constraint of S_1 is updated by projector (12.10) which is derived from propagation rule (11.18). The lower bounds of the cardinality constraints of S_2 and S_3 are updated by projectors (12.11) and (12.12) which are derived from propagation rule (11.19).

$$\lceil S_1 \rceil \dot{\subseteq} \lceil S_2 \rceil \cup \lceil S_3 \rceil \quad (12.7) \qquad \overline{|S_1|} \dot{\leq} \overline{|S_2|} + \overline{|S_3|} \quad (12.10)$$

$$\lfloor S_2 \rfloor \dot{\supseteq} \lfloor S_1 \rfloor \setminus \lceil S_3 \rceil \quad (12.8) \qquad \underline{|S_2|} \dot{\geq} \underline{|S_1|} - \overline{|S_3|} \quad (12.11)$$

$$\lfloor S_3 \rfloor \dot{\supseteq} \lfloor S_1 \rfloor \setminus \lceil S_2 \rceil \quad (12.9) \qquad \underline{|S_3|} \dot{\geq} \underline{|S_1|} - \overline{|S_2|} \quad (12.12)$$

The set of projectors for the constraint $S_1 \subseteq S_2 \cup S_3$ is denoted by $\mathcal{P}_{S_1 \subseteq S_2 \cup S_3}(S_1, S_2, S_3)$ where the parameters are accordingly replaced.

Constraint Projector Sets The set of constraint projectors for a conjunction of $S_1 \supseteq S_2 \cap S_3$ and $S_1 \subseteq S_2 \cup S_3$ constraints is computed by applying the rules (12.13) and (12.14).

$$\frac{S_1 \supseteq S_2 \cap S_3 \wedge C, \mathcal{P}}{C, \mathcal{P}_{S_1 \supseteq S_2 \cap S_3}(S_1, S_2, S_3) \cup \mathcal{P}} \quad (12.13) \qquad \frac{S_1 \subseteq S_2 \cup S_3 \wedge C, \mathcal{P}}{C, \mathcal{P}_{S_1 \subseteq S_2 \cup S_3}(S_1, S_2, S_3) \cup \mathcal{P}} \quad (12.14)$$

Normal Form of a Constraint Projector Set A constraint projector set $\mathcal{P}$ is in normal form if

1. for every projection variable S *resp.* $|S|$ of a projector there is at most one constraint projector of the form $\lfloor S \rfloor \dot{\supseteq} E^S$ and $\lceil S \rceil \dot{\subseteq} E^S$ *resp.* $\underline{|S|} \dot{\geq} E^C$ and $\overline{|S|} \dot{\leq} E^C$.
2. in set and cardinality expressions on the right hand-side of projectors every access function is only applied to a set variable S *resp.* cardinality variable $|S|$, *i.e.* only these applications $\lfloor S \rfloor$, $\lceil S \rceil$, $\underline{|S|}$ and $\overline{|S|}$ are allowed .

A normal form of a set of projections $\mathcal{P}$ is computed in two steps. At step 1, the rules (12.15)–(12.18) are applied to $\mathcal{P}$ to enforce condition 1. The rules (12.15) and (12.16) join two projectors to bound sets in the same fashion as the propagation rules (11.3) and (11.4) join basic bound constraints. Analogously, rules (12.17) and (12.18) correspond to the propagation rules (11.6) and (11.7).

$$\frac{\{\lfloor S\rfloor \dot{\supseteq} E_1^S, \lfloor S\rfloor \dot{\supseteq} E_2^S\} \uplus \mathcal{P}}{\{\lfloor S\rfloor \dot{\supseteq} E_1^S \cup E_2^S\} \uplus \mathcal{P}} \quad (12.15) \qquad \frac{\{\underline{|S|} \dot{\geq} E_1^C, \underline{|S|} \dot{\geq} E_2^C\} \uplus \mathcal{P}}{\{\underline{|S|} \dot{\geq} \max(E_1^C, E_2^C)\} \uplus \mathcal{P}} \quad (12.17)$$

$$\frac{\{\lceil S\rceil \dot{\subseteq} E_1^S, \lceil S\rceil \dot{\subseteq} E_2^S\} \uplus \mathcal{P}}{\{\lceil S\rceil \dot{\subseteq} E_1^S \cap E_2^S\} \uplus \mathcal{P}} \quad (12.16) \qquad \frac{\{\overline{|S|} \dot{\leq} E_1^C, \overline{|S|} \dot{\leq} E_2^C\} \uplus \mathcal{P}}{\{\overline{|S|} \dot{\leq} \min(E_1^C, E_2^C)\} \uplus \mathcal{P}} \quad (12.18)$$

Step 2 moves in the set expressions and cardinality expressions of all projectors $p \in \mathcal{P}$ the applications of the access functions to the variables to meet condition 2. The rules in Figure 12.1 take care of set expressions E^S. The rules (12.19) and (12.23) remove access function applications from set values.

$$\frac{\{\lfloor d\rfloor\} \uplus \mathcal{P}}{\{d\} \uplus \mathcal{P}} \quad (12.19) \qquad \frac{\{\lceil d\rceil\} \uplus \mathcal{P}}{\{d\} \uplus \mathcal{P}} \quad (12.23)$$

$$\frac{\{\lfloor E_1^S \setminus E_2^S\rfloor\} \uplus \mathcal{P}}{\{\lfloor E_1^S\rfloor \setminus \lceil E_2^S\rceil\} \uplus \mathcal{P}} \quad (12.20) \qquad \frac{\{\lceil E_1^S \setminus E_2^S\rceil\} \uplus \mathcal{P}}{\{\lceil E_1^S\rceil \setminus \lfloor E_2^S\rfloor\} \uplus \mathcal{P}} \quad (12.24)$$

$$\frac{\{\lfloor E_1^S \cap E_2^S\rfloor\} \uplus \mathcal{P}}{\{\lfloor E_1^S\rfloor \cap \lfloor E_2^S\rfloor\} \uplus \mathcal{P}} \quad (12.21) \qquad \frac{\{\lceil E_1^S \cap E_2^S\rceil\} \uplus \mathcal{P}}{\{\lceil E_1^S\rceil \cap \lceil E_2^S\rceil\} \uplus \mathcal{P}} \quad (12.25)$$

$$\frac{\{\lfloor E_1^S \cup E_2^S\rfloor\} \uplus \mathcal{P}}{\{\lfloor E_1^S\rfloor \cup \lfloor E_2^S\rfloor\} \uplus \mathcal{P}} \quad (12.22) \qquad \frac{\{\lceil E_1^S \cup E_2^S\rceil\} \uplus \mathcal{P}}{\{\lceil E_1^S\rceil \cup \lceil E_2^S\rceil\} \uplus \mathcal{P}} \quad (12.26)$$

Figure 12.1: Normalization rules for set expressions.

All rules not explicitly mentioned move access function applications to the set variables.

The rules in Figure 12.2 take care of cardinality expressions E^C. The rules (12.27) and (12.33) remove access function applications from integers. Note that rules (12.28) and (12.34) branch to the set of rules for normalizing set expressions.

All rules not explicitly mentioned move access function applications to cardinality of set variables.

If none of the rules (12.19)–(12.38) can be applied, for any constraint projector $p \in \mathcal{P}$, $\mathcal{P}$ is in normal form.

12.2 Computation of Events

Applying a propagation projector $p : S \odot E$ with $\odot \in \{\dot{\subseteq}, \dot{\supseteq}, \dot{\leq}, \dot{\geq}\}$ causes events on the projection variable S. These events are called *propagation events* and denoted by $\mathcal{E}_{\triangleleft}(S, \odot)$ (or for short $\mathcal{E}_{\triangleleft}(p)$).

A projector $p : S \odot E$ with $\odot \in \{\dot{\subseteq}, \dot{\supseteq}, \dot{\leq}, \dot{\geq}\}$ has to be re-executed if certain events on variables of its right hand-side expression ($\mathcal{V}(E)$) occur. These events are called *re-execution events* and are denoted by $\mathcal{E}_{\triangleright}(\odot, E)$ (or for short $\mathcal{E}_{\triangleright}(p)$).

$$\frac{\{\underline{n}\} \uplus \mathcal{P}}{\{n\} \uplus \mathcal{P}} \quad (12.27) \qquad \frac{\{\overline{n}\} \uplus \mathcal{P}}{\{n\} \uplus \mathcal{P}} \quad (12.33)$$

$$\frac{\{\underline{|E^S|}\} \uplus \mathcal{P}}{\{\underline{|\lfloor E^S \rfloor|}\} \uplus \mathcal{P}} \quad (12.28) \qquad \frac{\{\overline{|E^S|}\} \uplus \mathcal{P}}{\{\underline{|\lceil E^S \rceil|}\} \uplus \mathcal{P}} \quad (12.34)$$

$$\frac{\{\underline{|E_1^C + E_2^C|}\} \uplus \mathcal{P}}{\{\underline{|E_1^C|} + \underline{|E_2^C|}\} \uplus \mathcal{P}} \quad (12.29) \qquad \frac{\{\overline{|E_1^C + E_2^C|}\} \uplus \mathcal{P}}{\{\overline{|E_1^C|} + \overline{|E_2^C|}\} \uplus \mathcal{P}} \quad (12.35)$$

$$\frac{\{\underline{|E_1^C - E_2^C|}\} \uplus \mathcal{P}}{\{\underline{|E_1^C|} - \overline{|E_2^C|}\} \uplus \mathcal{P}} \quad (12.30) \qquad \frac{\{\overline{|E_1^C - E_2^C|}\} \uplus \mathcal{P}}{\{\overline{|E_1^C|} - \underline{|E_2^C|}\} \uplus \mathcal{P}} \quad (12.36)$$

$$\frac{\{\underline{\min(|E_1^C|, |E_2^C|)}\} \uplus \mathcal{P}}{\{\min(\underline{|E_1^C|}, \underline{|E_2^C|})\} \uplus \mathcal{P}} \quad (12.31) \qquad \frac{\{\overline{\min(|E_1^C|, |E_2^C|)}\} \uplus \mathcal{P}}{\{\min(\underline{|E_1^C|}, \underline{|E_2^C|})\} \uplus \mathcal{P}} \quad (12.37)$$

$$\frac{\{\underline{\max(|E_1^C|, |E_2^C|)}\} \uplus \mathcal{P}}{\{\max(\underline{|E_1^C|}, \underline{|E_2^C|})\} \uplus \mathcal{P}} \quad (12.32) \qquad \frac{\{\overline{\max(|E_1^C|, |E_2^C|)}\} \uplus \mathcal{P}}{\{\max(\underline{|E_1^C|}, \underline{|E_2^C|})\} \uplus \mathcal{P}} \quad (12.38)$$

Figure 12.2: Normalization rules for cardinality expressions.

Possible Events An event denotes a certain kind of update to a variable. The following events are defined for a variable S *resp.* $|S|$: update the lower bound set (denoted by $S^{e(\lfloor\cdot\rfloor)}$), update the upper bound set (denoted by $S^{e(\lceil\cdot\rceil)}$), update the lower bound of the cardinality (denoted by $S^{e(\underline{|\cdot|})}$), and update the upper bound of the cardinality (denoted by $S^{e(\overline{|\cdot|})}$).

Propagation Event Sets A propagation event set defines possible events caused on a projection variable by applying a propagation projector. The following propagation event sets are defined.

$$\mathcal{E}_{\lhd}(S, \dot{\subseteq}) = \{S^{e(\lceil\cdot\rceil)}, S^{e(\overline{|\cdot|})}, S^{e(\lfloor\cdot\rfloor)}\} \quad (12.39)$$

$$\mathcal{E}_{\lhd}(S, \dot{\supseteq}) = \{S^{e(\lfloor\cdot\rfloor)}, S^{e(\underline{|\cdot|})}, S^{e(\lceil\cdot\rceil)}\} \quad (12.40)$$

$$\mathcal{E}_{\lhd}(S, \dot{\leq}) = \{S^{e(\overline{|\cdot|})}, S^{e(\lceil\cdot\rceil)}\} \quad (12.41)$$

$$\mathcal{E}_{\lhd}(S, \dot{\geq}) = \{S^{e(\underline{|\cdot|})}, S^{e(\lfloor\cdot\rfloor)}\} \quad (12.42)$$

The events $S^{e(\overline{|\cdot|})}$ and $S^{e(\underline{|\cdot|})}$ in sets (12.39) and (12.40) are due to the invariants maintained by the propagation rules (11.10) and (11.11) on page 103. Event $S^{e(\lceil\cdot\rceil)}$ in the sets (12.40) and (12.41) is caused by propagation rule (11.8) while event $S^{e(\lfloor\cdot\rfloor)}$ in the sets (12.39) and (12.42) is caused by propagation rule (11.9). (See page 103 for the rules (11.8)–(11.11).)

Note that a propagation event set $\mathcal{E}_{\lhd}(S, \odot)$ with $\odot \in \{\dot{\subseteq}, \dot{\supseteq}, \dot{\leq}, \dot{\geq}\}$ contains only events on S.

Re-execution Event Sets A re-execution event set of an expression (a set expression E^S or a cardinality expression E^C) contains those events which may have an impact on the outcome of the evaluation of the expression. Hence, the events in a re-execution event set trigger the re-execution of the respective expression.

A set of events is denoted by $\mathcal{E}$. A set of re-execution events collects events which either increase or decrease the valuation of the expression. The valuation of a set expression E^S increases if $\text{val}(E^S) \subseteq \text{val}(E^S)^{\mathcal{E}}$ where $\text{val}(E^S)$ is the valuation of E^S before and $\text{val}(E^S)^{\mathcal{E}}$ is the valuation of E^S after an event $e \in \mathcal{E}$ occurred. Analogously, The valuation of a set expression E^S decreases if $\text{val}(E^S) \supseteq \text{val}(E^S)^{\mathcal{E}}$.

The projector $\lceil S \rceil \dot{\subseteq} E^S$ updates S in case of events on $\mathcal{V}(E^S)$ which decrease the valuation of E^S (equation (12.43)) while the projector $\lfloor S \rfloor \dot{\supseteq} E^S$ updates S in case of events on $\mathcal{V}(E^S)$ which increase the valuation of E^S (equation (12.44)).

$$\mathcal{E}_{\triangleright}(\dot{\subseteq}, E^S) = \mathcal{E} \downarrow E^S \qquad (12.43) \qquad\qquad \mathcal{E}_{\triangleright}(\dot{\supseteq}, E^S) = \mathcal{E} \uparrow E^S \qquad (12.44)$$

The actual re-execution event set for a set expression E^S is recursively described by the rules in Figure 12.3. Rules (12.45)–(12.47) and (12.51)–(12.53) define the events having an impact on variables and constants while rules (12.48)–(12.50) and(12.54)–(12.56) compose the event set according to the valuation and the possible set operators ($\cup$, $\cap$ and $\setminus$).

$$\emptyset \uparrow d \qquad (12.45) \qquad\qquad \emptyset \downarrow d \qquad (12.51)$$

$$\{S^{e(\lfloor \cdot \rfloor)}\} \uparrow \lfloor S \rfloor \qquad (12.46) \qquad\qquad \emptyset \downarrow \lfloor S \rfloor \qquad (12.52)$$

$$\{S^{e(\lceil \cdot \rceil)}\} \downarrow \lceil S \rceil \qquad (12.47) \qquad\qquad \emptyset \uparrow \lceil S \rceil \qquad (12.53)$$

$$\frac{\mathcal{E}_1 \uparrow E_1^S \quad \mathcal{E}_2 \uparrow E_2^S}{\mathcal{E}_1 \cup \mathcal{E}_2 \uparrow E_1^S \cup E_2^S} \qquad (12.48) \qquad\qquad \frac{\mathcal{E}_1 \downarrow E_1^S \quad \mathcal{E}_2 \downarrow E_2^S}{\mathcal{E}_1 \cup \mathcal{E}_2 \downarrow E_1^S \cup E_2^S} \qquad (12.54)$$

$$\frac{\mathcal{E}_1 \uparrow E_1^S \quad \mathcal{E}_2 \uparrow E_2^S}{\mathcal{E}_1 \cup \mathcal{E}_2 \uparrow E_1^S \cap E_2^S} \qquad (12.49) \qquad\qquad \frac{\mathcal{E}_1 \downarrow E_1^S \quad \mathcal{E}_2 \downarrow E_2^S}{\mathcal{E}_1 \cup \mathcal{E}_2 \downarrow E_1^S \cap E_2^S} \qquad (12.55)$$

$$\frac{\mathcal{E}_1 \uparrow E_1^S \quad \mathcal{E}_2 \downarrow E_2^S}{\mathcal{E}_1 \cup \mathcal{E}_2 \uparrow E_1^S \setminus E_2^S} \qquad (12.50) \qquad\qquad \frac{\mathcal{E}_1 \downarrow E_1^S \quad \mathcal{E}_2 \uparrow E_2^S}{\mathcal{E}_1 \cup \mathcal{E}_2 \downarrow E_1^S \setminus E_2^S} \qquad (12.56)$$

Figure 12.3: Rules for deriving re-execution event sets for set expressions.

The re-execution event set for a cardinality expressions is defined analogously to set expressions. The valuation of a cardinality expression E^C increases if $\text{val}(E^C) \leq \text{val}(E^C)^{\mathcal{E}}$ where $\text{val}(E^C)$ is the valuation of E^C before and $\text{val}(E^C)^{\mathcal{E}}$ is the valuation of E^C after an event $e \in \mathcal{E}$ occurred. Analogously, the valuation of a cardinality expression E^C decreases if $\text{val}(E^C) \geq \text{val}(E^C)^{\mathcal{E}}$.

The projector $\overline{|S|} \dot{\leq} E^C$ updates S in case of events on $\mathcal{V}(E^C)$ which decrease the

valuation of E^C (equation (12.57)) while the projector $\underline{|S|} \dot{\geq} E^C$ updates S in case of events on $\mathcal{V}(E^C)$ which increase the valuation of E^C (equation (12.58)).

$$\mathcal{E}_{\triangleright}(\dot{\leq}, E^C) = \mathcal{E} \downarrow E^C \qquad (12.57) \qquad \mathcal{E}_{\triangleright}(\dot{\geq}, E^C) = \mathcal{E} \uparrow E^C \qquad (12.58)$$

Event sets for cardinality expressions are described in an analogous way as for set expressions by the rules in Figure 12.4. Rules (12.59)–(12.61) and (12.67)–(12.69) define the events having an impact on variables and constants while rules (12.63)–(12.66) and (12.71)–(12.74) compose the event set according to the valuation and the arithmetic operators and functions ($+$, $-$, min and max). Rules (12.62) and (12.70) branch to the rules in Figure 12.3 for computing the valuation of the set expression argument of the cardinality operator ($\left|E^S\right|$).

$$\emptyset \uparrow n \qquad (12.59) \qquad \emptyset \downarrow n \qquad (12.67)$$

$$\{S^{e(\underline{|\cdot|})}\} \uparrow \underline{|S|} \qquad (12.60) \qquad \emptyset \downarrow \underline{|S|} \qquad (12.68)$$

$$\{S^{e(\overline{|\cdot|})}\} \downarrow \overline{|S|} \qquad (12.61) \qquad \emptyset \uparrow \overline{|S|} \qquad (12.69)$$

$$\frac{\mathcal{E} \uparrow E^S}{\mathcal{E} \uparrow \left|E^S\right|} \qquad (12.62) \qquad \frac{\mathcal{E} \downarrow E^S}{\mathcal{E} \downarrow \left|E^S\right|} \qquad (12.70)$$

$$\frac{\mathcal{E}_1 \uparrow E_1^C \quad \mathcal{E}_2 \uparrow E_2^C}{\mathcal{E}_1 \cup \mathcal{E}_2 \uparrow E_1^C + E_2^C} \qquad (12.63) \qquad \frac{\mathcal{E}_1 \downarrow E_1^C \quad \mathcal{E}_2 \downarrow E_2^C}{\mathcal{E}_1 \cup \mathcal{E}_2 \downarrow E_1^C + E_2^C} \qquad (12.71)$$

$$\frac{\mathcal{E}_1 \uparrow E_1^C \quad \mathcal{E}_2 \downarrow E_2^C}{\mathcal{E}_1 \cup \mathcal{E}_2 \uparrow E_1^C - E_2^C} \qquad (12.64) \qquad \frac{\mathcal{E}_1 \downarrow E_1^C \quad \mathcal{E}_2 \uparrow E_2^C}{\mathcal{E}_1 \cup \mathcal{E}_2 \downarrow E_1^C - E_2^C} \qquad (12.72)$$

$$\frac{\mathcal{E}_1 \uparrow E_1^C \quad \mathcal{E}_2 \uparrow E_2^C}{\mathcal{E}_1 \cup \mathcal{E}_2 \uparrow \min(E_1^C, E_2^C)} \qquad (12.65) \qquad \frac{\mathcal{E}_1 \downarrow E_1^C \quad \mathcal{E}_2 \downarrow E_2^C}{\mathcal{E}_1 \cup \mathcal{E}_2 \downarrow \min(E_1^C, E_2^C)} \qquad (12.73)$$

$$\frac{\mathcal{E}_1 \uparrow E_1^C \quad \mathcal{E}_2 \uparrow E_2^C}{\mathcal{E}_1 \cup \mathcal{E}_2 \uparrow \max(E_1^C, E_2^C)} \qquad (12.66) \qquad \frac{\mathcal{E}_1 \downarrow E_1^C \quad \mathcal{E}_2 \downarrow E_2^C}{\mathcal{E}_1 \cup \mathcal{E}_2 \downarrow \max(E_1^C, E_2^C)} \qquad (12.74)$$

Figure 12.4: Rules for deriving re-execution event sets for cardinality expressions.

Note that a re-execution event set $\mathcal{E}_{\triangleright}(\odot, E)$ with $\odot \in \{\dot{\subseteq}, \dot{\supseteq}, \dot{\leq}, \dot{\geq}\}$ contains only events on $\mathcal{V}(E)$.

12.3 Filter Generation

The constraint projectors realizing a given constraint are arranged such that a the resulting filter is idempotent. But projectors typically depend on each other so that idempotent behavior is achieved by looping over mutually dependent constraint projectors and by

ordering projectors such that the dependencies and the execution order go in the same direction.

Constraint projectors are executed top-down, *i.e.*, the top projector p_{top} is executed first and the bottom projector p_{bottom} is executed last. The execution order is expressed by an order on projectors $p_{top} < \ldots < p_{bottom}$.

Dependent Constraint Projectors A constraint projector p_1 depends on a projector p_2 if $\mathcal{E}_{\triangleright}(p_1) \not\parallel \mathcal{E}_{\triangleleft}(p_2)$. In words, the set of propagation events $\mathcal{E}_{\triangleleft}(p_2)$ shares events with the set of re-execution events $\mathcal{E}_{\triangleright}(p_1)$ and the shared events $\mathcal{E}_{\triangleright}(p_1) \cap \mathcal{E}_{\triangleleft}(p_2)$ may have an impact on p_1.

Consider the example in Figure 12.5, where p_1 depends on p_2 because of the upwards dependency caused by the event $S_2^{e(\lceil\cdot\rceil)}$. Hence, p_1 and p_2 are re-ordered to execute p_2 before p_1 and thus, the upwards dependency is avoided.

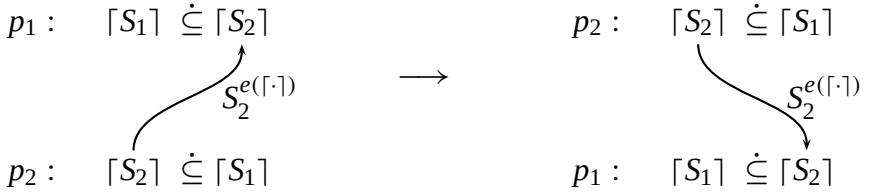


Figure 12.5: Resolving upwards-dependent constraint projectors by re-ordering.

Mutually dependent constraint projectors cannot be resolved by re-ordering. See the example in Figure 12.6. The projectors p_1 and p_2 are looped over until they reach a fixed-point, *i.e.*, no event $S_2^{e(\lceil\cdot\rceil)}$ occurs because p_2 does change $\lceil S_2 \rceil$ anymore.

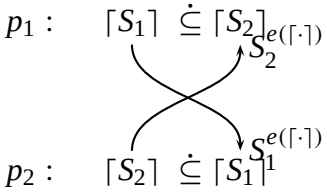


Figure 12.6: Mutally dependent constraint projectors.

Graph-based Analysis Figures 12.5 and 12.6 suggest to represents constraint projectors and the dependencies among them as graph. This graph is called *dependency graph* and denoted by $G = (N, E)$ where N is the set of nodes and E the set of directed edges. A node $n_p \in N$ represents a constraint projector p. A directed edge (n_{p_i}, n_{p_j}) denotes that projector p_j depends on updates of projector p_i. The set of edges is constructed by

$$E = \{(n_{p_i}, n_{p_j}) \mid \mathcal{E}_{\triangleleft}(p_i) \not\parallel \mathcal{E}_{\triangleright}(p_j)\} \tag{12.75}$$

The execution order on constraint projectors represented by a graph G is obtained by topologically sorting G. In case an topological order is found, the projectors are sequentially arranged according to that topological order. In case no order is found, *i.e.*, the dependency graph contains cycles, the set of strongly connected components $\mathcal{SCC}_G$ (see for example [34]) of G are computed. Projectors $\mathcal{P}_{scc}$ represented by a strongly connected component $\mathcal{S}_{scc} \in \mathcal{SCC}_G$ where $|\mathcal{S}_{scc}| > 1$ are mutually dependent with each other and are executed in a loop until a fixed-point is detected.

Generating Loops There is a loop for every strongly connected component $\mathcal{S}_{scc} \in \mathcal{SCC}_G$ where $|\mathcal{S}_{scc}| > 1$. The nodes of $\mathcal{S}_{scc}$ represent the projectors in $\mathcal{P}_{scc}$. It is desirable to find an order for the projectors in $\mathcal{P}_{scc}$ such that the number u of upwards dependencies is minimal because u is a measure for the complexity of the test for detecting the fixed-point. Unfortunately, finding the minimal u is NP-hard due to the underlying linear-ordering problem [57] (see Section 12.5 for a discussion). Because of the complexity for finding an optimal ordering, the projectors in $\mathcal{P}_{scc}$ are arbitrarily ordered $\forall p_1, \ldots, p_n \in \mathcal{P}_{scc} : p_1 < \ldots < p_n$.

Events and Profiles A loop has reached its fixed-point if no fixed-point events have occurred during an iteration. Fixed-point events state an upwards dependency between ordered projectors. The set of fixed-point events $\mathcal{E}_{fix}$ is computed by:

$$\mathcal{E}_{fix} = \bigcup_{p_i, p_j \in \mathcal{P} : p_i < p_j} \mathcal{E}_{\triangleright}(p_i) \cap \mathcal{E}_{\triangleleft}(p_j) \tag{12.76}$$

The occurrence of fixed-point events is detected using constraint profiles which are collected in the set of fixed-point profiles $\mathcal{T}_{fix}$. The set of fixed-point profiles $\mathcal{T}_{fix}$ is created by mapping fixed-point events $S^{e(\odot)} \in \mathcal{E}_{fix}$ to the corresponding profiles $S^{p(\odot)}$:

$$\mathcal{T}_{fix} = \{S^{p(\odot)} \mid S^{e(\odot)} \in \mathcal{E}_{fix}\} \tag{12.77}$$

At the beginning of a loop, all components of basic constraints which produce fixed-point events are profiled. At the end of the loop, the profiles are used to test if the respective component is unchanged, *i.e.* has not caused an fixed-point events. This is done by the predicate $S = S^{p(\odot \in \{\lfloor\cdot\rfloor, \lceil\cdot\rceil, \underline{\lfloor\cdot\rfloor}, \overline{\lceil\cdot\rceil}\})}$ which is true if the component $\odot$ to be tested is unchanged since the profile $S^{p(\odot)}$ has been taken. If no fixed-point event occurred, the loop is left. Looping is done by a modified **`repeat-until`**-loop:

`repeat`$[\mathcal{T}_{fix}$ **`]{`**
  $p_1 \in \mathcal{P}_{scc}$
  ...
  $p_n \in \mathcal{P}_{scc}$
`}` **`until`** $(\bigwedge_{S^{p(\odot)} \in \mathcal{T}_{fix}} S = S^{p(\odot)})$**`;`**

The semantics of this loop is as follows. Take at the beginning of every iteration every profile in $\mathcal{T}_{fix}$ ($\mathcal{T}_{fix}$ is computed for $\mathcal{P}_{scc}$). Then execute the projectors p_1 to p_n. Eventually, test if fixed-point events occurred by comparing the profiles with the respective components. The loop is left if no fixed-point events occurred. Else reiterate.

Generating a Sequential Algorithm After determining fixed-point loops, a graph G' is constructed where the nodes are the collapsed strongly connected components $\mathcal{S}_{scc} \in \mathcal{SCC}_G$ and the edges are those edges of G connecting the strongly connected components. The modified dependency graph G' is topologically sorted. Note that a node of G' denotes either a single constraint projector or a loop. The topological order of G' determines the sequential order of the constraint projectors and loops in the generated algorithm.

Compiling the Algorithm The algorithm for constructing filter algorithms is shown in Figure 12.7. It comprise all steps from the initial set constraint to be realized till the generated filter algorithm.

Integrating Filter Algorithms in Propagators Filters are straightforwardly integrated into propagators. The issues are failure, entailment and propagator scheduling. A generated filter detects automatically failure and entailment if all parameters are determined (conditions (adequate.1) and (adequate.2) on page 11). Failure is detected by the projectors if either rule (11.2) or (11.5) fires (conditions (adequate.1)). A propagator is entailed if its filter determines all its parameters and does not fail (conditions (adequate.2)). A propagator p has to be scheduled if any of the re-execution events of the projectors of its filter occur on its parameters. Propagator p schedules other propagators if F_p causes the respective events on its parameters. This functionality is encapsulated in the access variables defined for finite set variables.

12.4 An Example for Filter Construction

This section illustrates the algorithm in Figure 12.7 by constructing a filter algorithm performing bound propagation for the set expression $S_1 = S_2 \cap S_3$. Additionally, a filter algorithm performing bound and cardinality propagation for the same constraint is given.

A Filter for Bounds Propagation The expression $S_1 = S_2 \cap S_3$ can be straightforwardly rewritten to $S_1 \supseteq S_2 \cap S_3 \ \wedge \ S_1 \subseteq S_2 \ \wedge \ S_1 \subseteq S_3$ (see Figure 11.1). The corresponding set of constraint projectors $\mathcal{P}$ consists of the following projectors:

①	$\lfloor S_1 \rfloor \dot{\supseteq} \lfloor S_2 \rfloor \cap \lfloor S_3 \rfloor$	④	$\lceil S_1 \rceil \dot{\subseteq} \lceil S_2 \rceil \cap \lceil S_3 \rceil$
②	$\lfloor S_2 \rfloor \dot{\supseteq} \lfloor S_1 \rfloor$	⑤	$\lceil S_2 \rceil \dot{\subseteq} (\lfloor S_3 \rfloor \setminus \lceil S_1 \rceil)^{\complement}$
③	$\lfloor S_3 \rfloor \dot{\supseteq} \lfloor S_1 \rfloor$	⑥	$\lceil S_3 \rceil \dot{\subseteq} (\lfloor S_2 \rfloor \setminus \lceil S_1 \rceil)^{\complement}$

Note that $\lceil S_1 \rceil \dot{\subseteq} \lceil S_2 \rceil$ and $\lceil S_1 \rceil \dot{\subseteq} \lceil S_3 \rceil$ are collapsed to $\lceil S_1 \rceil \dot{\subseteq} \lceil S_2 \rceil \cap \lceil S_3 \rceil$ (see rule (12.16)).

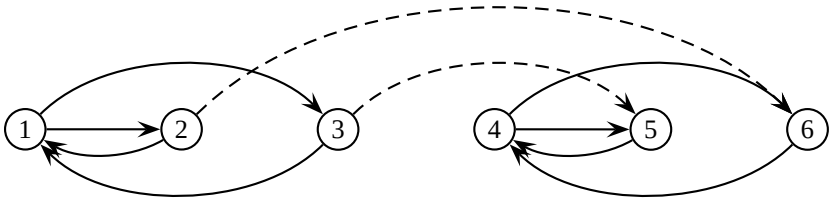
Next, the dependency graph is constructed from $\mathcal{P}$ and topologically sorted. Since there is no topological order, all strongly connected components are computed. The edges inside strongly connected components are solid while the other edges are dashed:

Input: a non-basic constraint C as defined in Section 11.2
Output: a filter algorithm

1. Expand C by the rules (12.13)–(12.14) to $\mathcal{P}$ and normalize $\mathcal{P}$ by the rules (12.15)–(12.38) to $\mathcal{P}'$.
2. Compute for every $p \in \mathcal{P}'$ the set of re-execution events $\mathcal{E}_{\triangleright}(p)$ and the set of propagation events $\mathcal{E}_{\triangleleft}(p)$.
3. Construct the dependency graph $G = (N, E)$ where N is the set of nodes representing the constraint projectors and E is the set of directed edges (see equation (12.75)) referring from a projector-node causing propagation events to the projector-node where these events have an impact (*i.e.*, the projector has to be re-executed).
4. Sort G topologically:
   (a) If a topological order is found then there are no cycles in G. Arrange the constraint projectors sequentially according to the topological order. Stop.
   (b) If *no* topological order is found then G contains cycles. Compute the set $\mathcal{SCC}_G$ of strongly connected components of G:
      i. Every element $\mathcal{S}_{scc} \in \mathcal{SCC}_G$ denotes a sequence of constraint projectors being subject to fixed-point iteration. Create a loop for every $\mathcal{S}_{scc} \in \mathcal{SCC}_G$. The fixed-point profiles are computed by mapping the fixed-point event set of $\mathcal{S}_{scc}$ to the corresponding profile set (see equations (12.76) and (12.77)).
      ii. Collapse all $\mathcal{S}_{scc} \in \mathcal{SCC}_G$ to single nodes and sort the collapsed dependency graph G' topologically. Arrange the loops and the constraint projectors sequentially according to the topological order. Stop.

Disconnected sub-graphs of the dependency graph can be arranged in any order.

Figure 12.7: Algorithm for the generation of filter algorithms.



The left strongly connected component $\mathcal{S}_{scc_A}$ contains the projectors ①, ②, and ③ while the right strongly connected component $\mathcal{S}_{scc_B}$ contains projectors ④, ⑤, and ⑥. Collapsing $\mathcal{S}_{scc_A}$ and $\mathcal{S}_{scc_B}$ to single nodes and topologically sorting the collapsed dependency

graph determines $\mathcal{S}_{scc_A}$ to be placed before $\mathcal{S}_{scc_B}$.

Finally, the fixed-point profile sets are computed. The projectors are arranged in the following execution order: $\mathcal{S}_{scc_A} : ② < ③ < ①$ and $\mathcal{S}_{scc_B} : ⑤ < ④ < ⑥$. The generated filter algorithm for the constraint $S_1 = S_2 \cap S_3$ is shown in Program 12.1.

$S_1 = S_2 \cap S_3$:
  repeat $[{S_1}^{p(\lfloor\cdot\rfloor)}]$ {
    $\lfloor S_2 \rfloor \dot{\supseteq} \lfloor S_1 \rfloor$;
    $\lfloor S_3 \rfloor \dot{\supseteq} \lfloor S_1 \rfloor$;
    $\lfloor S_1 \rfloor \dot{\supseteq} \lfloor S_2 \rfloor \cap \lfloor S_3 \rfloor$;
  } **until** $({S_1} = {S_1}^{p(\lfloor\cdot\rfloor)})$;
  repeat $[{S_1}^{p(\lceil\cdot\rceil)}]$ {
    $\lceil S_2 \rceil \dot{\subseteq} \lfloor S_3 \rfloor^{\complement} \cup \lceil S_1 \rceil$;
    $\lceil S_3 \rceil \dot{\subseteq} \lfloor S_2 \rfloor^{\complement} \cup \lceil S_1 \rceil$;
    $\lceil S_1 \rceil \dot{\subseteq} \lceil S_2 \rceil \cap \lceil S_3 \rceil$;
  } **until** $({S_1} = {S_1}^{p(\lceil\cdot\rceil)})$;

Program 12.1: Filter algorithm for $S_1 = S_2 \cap S_3$ with bounds reasoning.

A Filter for Bounds and Cardinality Propagation Adding cardinality propagation produces a dependency graph being a single strongly connected component. This results in a filter algorithm consisting of a single iteration loop as shown in Program 12.2.

$S_1 = S_2 \cap S_3$:
  repeat $[{S_3}^{p(\overline{|\cdot|})}, {S_1}^{p(\lfloor\cdot\rfloor)}, {S_1}^{p(\lceil\cdot\rceil)}, {S_3}^{p(\lfloor\cdot\rfloor)}, {S_1}^{p(\underline{|\cdot|})}, {S_3}^{p(\underline{|\cdot|})}, {S_1}^{p(\overline{|\cdot|})}, {S_3}^{p(\lceil\cdot\rceil)}, {S_2}^{p(\lceil\cdot\rceil)}]$ {
    $\overline{|S_2|} \dot{\leq} |\lceil S_2 \rceil \cup \lceil S_3 \rceil| + \overline{|S_1|} - \underline{|S_3|}$;
    $\underline{|S_2|} \dot{\geq} \underline{|S_1|}$;
    $\lceil S_2 \rceil \dot{\subseteq} \lfloor S_3 \rfloor^{\complement} \cup \lceil S_1 \rceil$;
    $\lfloor S_2 \rfloor \dot{\supseteq} \lfloor S_1 \rfloor$;
    $\overline{|S_1|} \dot{\leq} \min(\overline{|S_2|}, \overline{|S_3|})$;
    $\underline{|S_1|} \dot{\geq} \underline{|S_2|} + \underline{|S_3|} - |\lceil S_2 \rceil \cup \lceil S_3 \rceil|$;
    $\lceil S_1 \rceil \dot{\subseteq} \lceil S_2 \rceil \cap \lceil S_3 \rceil$;
    $\lfloor S_1 \rfloor \dot{\supseteq} \lfloor S_2 \rfloor \cap \lfloor S_3 \rfloor$;
    $\overline{|S_3|} \dot{\leq} |\lceil S_2 \rceil \cup \lceil S_3 \rceil| + \overline{|S_1|} - \underline{|S_2|}$;
    $\underline{|S_3|} \dot{\geq} \underline{|S_1|}$;
    $\lceil S_3 \rceil \dot{\subseteq} \lfloor S_2 \rfloor^{\complement} \cup \lceil S_1 \rceil$;
    $\lfloor S_3 \rfloor \dot{\supseteq} \lfloor S_1 \rfloor$;
  } **until** $({S_3} = {S_3}^{p(\overline{|\cdot|})} \wedge {S_1} = {S_1}^{p(\lfloor\cdot\rfloor)} \wedge {S_1} = {S_1}^{p(\lceil\cdot\rceil)}$
    $\wedge {S_3} = {S_3}^{p(\lfloor\cdot\rfloor)} \wedge {S_1} = {S_1}^{p(\underline{|\cdot|})} \wedge {S_3} = {S_3}^{p(\underline{|\cdot|})}$
    $\wedge {S_1} = {S_1}^{p(\overline{|\cdot|})} \wedge {S_3} = {S_3}^{p(\lceil\cdot\rceil)} \wedge {S_2} = {S_2}^{p(\lceil\cdot\rceil)})$;

Program 12.2: Filter algorithm for $S_1 = S_2 \cap S_3$ with bounds and cardinality reasoning.

12.5 Discussion

The implementation of the filter generator orders projectors to minimize the complexity of the detection of loop fixed-points. This is done in a naïve way by testing all permutations up to a certain number of projectors. Beyond this number no ordering is done since finding an optimal solution of the underlying linear-ordering problem is NP-hard [57]. Otherwise, the obtained benefit of the optimization is not clear due to the computational cost of projectors within the loops.

Filter generation compiles constraints to projectors. The idea of compiling constraints has previously been used in indexical-based finite domain solvers as discussed by Codognet and Diaz [32] and Carlson and Carlsson [21, Section 3]. Like projectors, indexicals compute basic constraints and write them to the constraint store. But indexicals are submitted to a run-time system which takes care of computing a fixed-point by using a scheduling mechanism similar to the one use for propagators in Part I. Hence, no analysis during compilation of indexicals is needed for ensuring a propagation fixed-point.

Georget and Codognet encode in [53] global constraints by set-based semi-rings (employing the semi-ring constraint solving framework presented in [17, 18, 16]). It is not obvious if the filter discussed in this chapter are covered by the semi-ring based constraint solving framework.

Chapter 13

Programming with Finite Integer Sets in Mozart

This chapter demonstrates finite integer set constraint programming in Mozart. It introduces the finite integer set library and illustrates problem solving with finite set constraints by several case studies. Furthermore, the connection with finite domain constraints for attributing individual elements of sets and breaking symmetry is discussed. This chapter is closed with a performance evaluation and a discussion of related work.

13.1 The Finite Integer Set Constraint Library

Finite integer set constraints are integrated into Mozart Oz by the *finite integer set constraint library* (short set library). This section gives an overview over the provided functionality and explains abstractions used later in this chapter. This library can be divided into three parts: basic constraints (Section 13.1.1), propagators, and distribution (Section 13.1.4). The discussion of propagators covers standard set relations (Section 13.1.2) and propagators connecting finite set with finite domain constraints (Section 13.1.3). A complete reference to the finite integer set constraint library can be found in Section 7 in [46].

13.1.1 Imposing and Reflecting Basic Constraints

Basic constraints can be imposed by the abstractions shown the following table:

abstraction	**basic constraint**
`{FS.var.lowerBound S D}`	$d \subseteq S$
`{FS.var.upperBound S D}`	$S \subseteq d$
`{FS.cardRange N1 N2 S}`	$n1 \leq \|S\| \leq n2$

`D` is bound to an Oz term denoting a *setdescr* which represents a set of integers (see Figure 3.3 on page 18 for details). `FS.var.lowerBound` and `FS.var.upperBound` synchronize on `D` to be determined and `FS.cardRange` synchronizes on `N1` and `N2` to be determined to integer.

A determined set variable can be created by `{FS.value.make` *setdescr* `S}` which constrains `S` to the set of integers described by *setdescr* and synchronizes on *setdescr*. Frequently occurring sets as the empty set ($\emptyset$) and the universal set ($\mathcal{U}$) are provided by the value constants `FS.value.empty` and `FS.value.universal`, respectively.

Reflection of Basic Constraints The reflection of basic constraints is done by the library abstraction shown in the following table:

abstraction	reflected basic constraint
`{FS.reflect.lowerBound S D}`	$d \subseteq S$
`{FS.reflect.upperBound S D}`	$S \subseteq d$
`{FS.reflect.unknown S D}`	$d_1 \subseteq S \subseteq d_2 \wedge d = d_2 \setminus d_1$

The abstractions above bind `D` to a compact[1] *setdescr* value denoting the corresponding set of integers.

The basic cardinality constraint $N1 \leq |S| \leq N2$ can be reflected by the abstraction `{FS.reflect.card S R}` where `R` is bound to `R=N1#N2`.

Note that the result of all reflection abstractions is dependent on the current state of the constraint store.

13.1.2 Propagators for Standard Set Operators

The set library provides a comprehensive set of propagators for set operators. Such a propagator can be nested, *i.e.*, it synchronizes on $n-1$ of its n parameters to be constrained by a finite integer set constraint. The parameter, say x, left unconstrained is automatically constraint to $\emptyset \subseteq x \subseteq \mathcal{U}$. The following table shows some of the propagators provided:

propagator	set operator
`{FS.intersect X Y Z}`	$\mathtt{Z} = \mathtt{X} \cap \mathtt{Y}$
`{FS.union X Y Z}`	$\mathtt{Z} = \mathtt{X} \cup \mathtt{Y}$
`{FS.diff X Y Z}`	$\mathtt{Z} = \mathtt{X} \setminus \mathtt{Y}$
`{FS.disjoint X Y}`	$\mathtt{X} \parallel \mathtt{Y}$
`{FS.subseteq X Y}`	$\mathtt{X} \subseteq \mathtt{Y}$

Generic Propagators Many applications require to have n-ary versions of the standard set operators. Such *generic propagators* make it possible to determine the number of parameters at run-time. This library provides for various set operators generic versions.

propagator	set operator
`{FS.intersectN` $\langle \mathtt{X}_1 \ldots \mathtt{X}_n \rangle$ `Z}`	$\mathtt{Z} = \cap_{i=1}^{n} \mathtt{X}_i$
`{FS.unionN` $\langle \mathtt{X}_1 \ldots \mathtt{X}_n \rangle$ `Z}`	$\mathtt{Z} = \cup_{i=1}^{n} \mathtt{X}_i$
`{FS.disjointN` $\langle \mathtt{X}_1 \ldots \mathtt{X}_n \rangle$`}`	$i, j \in \{1, \ldots, n\} : \mathtt{X}_{i, i \neq j} \parallel \mathtt{X}_j$
`{FS.partition` $\langle \mathtt{X}_1 \ldots \mathtt{X}_n \rangle$ `Z}`	$\mathtt{Z} = \uplus_{i=1}^{n} \mathtt{X}_i$

[1] Suppose the set $\{1, 2, 3, 4, 7\}$. A compact list representation is `[1#4 7]`. There are corresponding reflection abstractions returning an expanded representation (`[1 2 3 4 7]`) which are useful for programming iterations over individual elements of a set.

All propagators for standard set operators perform cardinality reasoning based on the rules given in Section 11.2.

13.1.3 Connecting Finite Domain and Finite Integer Sets

This section shows how the constraints proposed in Section 11.3 and some extensions are provided in the finite integer set library.

Membership and Cardinality The abstraction `{FS.include I S}` imposes the constraint $I \in S$ while `{FS.exclude I S}` imposes $I \notin S$. The abstraction `{FS.card I S}` imposes the cardinality constraint $\mathtt{I} = |\mathtt{S}|$. All three propagators are nestable.

Integer Domain as Ordered Domain The set library provides a variety of propagators for constraints that take the integer domain as an ordered domain. The most prominent of them are presented.

The propagator `{FS.int.match S` $\langle \mathtt{I}_1 \ldots \mathtt{I}_n \rangle$`}` implements the match-constraint $S = \{I_1, \ldots, I_n\}$. It connects the elements of `S` with finite domain variables in a vector (Section 13.2.1). The propagators `{FS.int.min S I}` and `{FS.int.max S I}` constrain `I` to be the minimum *resp.* maximum element of `S`.

The propagator `{FS.int.convex S}` constrains the set S to contain for $I_A, I_B \in S$ all I between I_A and I_B. The sequence propagator `{FS.int.seq` $\langle \mathtt{S}_1 \ldots \mathtt{S}_n \rangle$`}` imposes an order on sets in a vector: $\forall i, j \in \{1, \ldots, n\}, i < j : \forall m \in \mathtt{S}_i, \forall n \in \mathtt{S}_j : m < n$. These two propagators are particularly useful in linguistic applications (see Section 13.2.3).

Reification of Membership Reified constraints can be used to bind a $0/1$-variable to 1 (0) when a constraint is entailed (dis-entailed). Otherwise, reification can be used to determine at run-time whether a constraint c or its complement $\neg c$ is eventually imposed. For that purpose, the propagator `{FS.reified.include I S J}` is provided which implements the constraint $I \in S \leftrightarrow J$.

The propagator `{FS.reified.isIn I S J}` implements *directed reification* of $\mathtt{I} \in \mathtt{S}$, *i.e.* $\mathtt{I} \in \mathtt{S} \rightarrow \mathtt{J}$, which does not impose $\mathtt{I} \in \mathtt{S}$ or $\mathtt{I} \notin \mathtt{S}$ if `J` is bound.

The propagator `{FS.reified.areIn` $\langle \mathtt{I}_1 \ldots \mathtt{I}_n \rangle$ `S` $\langle \mathtt{J}_1 \ldots \mathtt{J}_n \rangle$`}` is added for the programmer's convenience and imposes `{FS.reified.isIn` $\mathtt{I}_i$ `S` $\mathtt{J}_i$`}` for all $i \in \{1, \ldots, n\}$.

13.1.4 Distribution

The implementation of branching and exploration algorithms (see Section 2.1) in finite set constraint programs is supported by the abstraction

`{FS.distribute Strategy` $\langle \mathtt{S}_1 \ldots \mathtt{S}_n \rangle$`}`

which is highly configurable. The first argument `Strategy` determines the branching and exploration algorithm used while the second argument is a vector of finite integer set variables to distribute on.

The most straightforward distribution strategy is called naïve (`Strategy=naive`) which takes the left-most variable S_1 with $d_1 \subseteq S \subseteq d_2$ of the vector and creates the disjunction $n \in S_1 \ \vee \ n \notin S_1$ where $n = \min(d_2 \setminus d_1)$ and $\min(S)$ returns the smallest integer of S (see Figure 11.2 on page 109 for an example). It distributes a variable until it is determined and then moves on to the next variable in `S`.

The value of `Strategy` may control the selection of the next variable and next element taken from this variable to distribute on. This selection process may additionally take into some attribution of the elements. There are for most common cases predefined functioned strategies but `FS.distribute` is flexible enough to accept also self-defined abstractions for new strategies.

Further, it can be controlled that a selected variable is either completely distributed before moving on to the next variable or that a new variable is selected for every distribution step. An application of this feature is demonstrated in Section 13.2.2 for implementing a round-robin distribution strategy.

It is possible to pass a procedure to `FS.distribute` which is applied as soon as propagation reaches its fixed-point. The default procedure is **`proc {$} skip end`**. This feature is particularly useful for programming with first-class constraints as discussed in Section 14.3.

13.1.5 Implementation Aspects

The implementation of the finite integer set solver consists of three parts: basic constraints, propagators and distribution as discussed in the following.

Basic Constraints Basic finite integer set constraints are fully integrated in the virtual machine of Mozart which is justified by the obtained gain in efficiency (Section 10.3). A finite integer set variable is derived from a constraint variable (Section 7.2.1). The attached event lists make it possible to trigger propagators if the bounds of the set constraint have been constrained *resp.* the constraint is constrained to denote a set value. The lower and upper bound finite integer sets of the actual basic constraint (Section 11.1) are implemented as described in Section 8.2. The cardinality constraint is represented by a pair of integers denoting the lower and upper bound.

The unification procedure of the VM is extended for finite integer set variables following the rules (11.3–11.2) on page 102. The library abstractions for imposing and reflecting basic constraints are implemented by foreign functions (Section 5.1).

Propagator All set propagators are implemented via the CPI (Chapter 8). The CPI is extended by classes for set values, set constraints, and access variable for set constraints, to make the implementation of set propagators possible. The filter of the propagators for the standard set operators are based on filter algorithms generated with the scheme presented in Chapter 12. They perform bounds and cardinality reasoning.

Distribution The library abstraction for distribution (`FS.distribute`) is completely implemented in Oz itself. It uses finite domain distribution in conjunction with the reified membership propagator `FS.reified.include`.

13.2 Case Studies

This section discusses three case studies for programming with finite integer set constraints in Mozart Oz. The first one, the ternary Steiner problem, has its origin in combinatorial mathematics. It is used to demonstrate how symmetry breaking constraints for finite sets can be implemented and what effect they may have (see also Section 13.3 on that). The second case study computes a schedule for a golf tournament. It turned out that only set constraint formulations of the problem are able to compute tournament schedules for a period of nine weeks. This example is also used to demonstrate how individual elements of a set can be attributed. The last case study discusses the computer linguistic application of dependency parsing. This application heavily benefits from Mozart's set propagators over the integer domain.

13.2.1 The Ternary Steiner Problem

The ternary Steiner problem belongs to a class of block theory problems from combinatorial mathematics which deal with hyper-graphs[2]. Beldiceanu discussed the problem the first time in computer science in [10]. A ternary Steiner problem of order n asks for $n(n-1)/6$ sets the $s_i \subset \{1,\ldots,n\}$ with cardinality 3 such that every two of them share at most one element. It has been proved that for a solution to exist n *mod* 6 must be 1 or 3.

Function `SteinerConstraints` creates first a list of $n(n-1)/6$ variables `Ss`.

```
fun {SteinerConstraints N} Ss = {MakeList (N*(N-1)) div 6} in
```

Next the variables $\mathtt{S}_i$ of `Ss` are constrained to $\emptyset \subseteq \mathtt{S}_i \subseteq \{1,\ldots,n\}$ and $|\mathtt{S}_i| = 3$.

```
   {ForAll Ss
    proc {$ S} {FS.var.upperBound 1#N S}  {FS.card S 3} end}
```

The nested loops `ForAllTail` and `ForAll` impose on all distinct pairs of $\mathtt{S}_i, \mathtt{S}_j \in \mathtt{Ss}$ the constraints $0 \leq \left|(\mathtt{S}_i \cap \mathtt{S}_j)\right| \leq 1$. Finally, the problem variable `Ss` is returned.

```
   {ForAllTail Ss proc {$ S1|Sr}
                     {ForAll Sr proc {$ S2} S3 in
                                   {FS.intersect S1 S2 S3}
                                   {FS.cardRange 0 1 S3}
                                end}
                  end}
   Ss
end
```

The function `Steiner` below maps an integer `N` to an (anonymous) procedure which models the Steiner problem of order n. Procedure `SteinerConstraints` imposes the constraints for the Steiner Problem. Finally, the distribution strategy is specified to always pick the leftmost undetermined `S` of `Ss` and the smallest integer n of the undecided set d of `S`, and then to distribute the choice point $n \in S \vee n \notin S$ where $n \min(d)$ (similar to Figure 11.2).

[2] A hyper-graph is a graph where edges may connect more than two nodes.

```
fun {Steiner N}
   proc {$ Ss}
      {SteinerConstraints N Ss}
      {FS.distribute naive Ss}
   end
end
```

For example, to solve the Steiner problem of order 9, one may invoke Mozart Oz's single solution search engine by executing `{SearchOne {Steiner 9} Sol}` which binds the solution to `Sol`.

Ordering Sets to Break Symmetries

Problem formulations asking for a collection of sets run into the risk of having numerous symmetric solutions. This can be avoided if an order on sets is available. Such an order can, for example be given in terms of an integer rank $rank(s)$ associated with every set s.

An immediate way of defining a rank is to interpret the characteristic function of every set as a bit string *resp.* as a binary number.

$$(b_0, b_1, \ldots, b_{\sup})_2 \quad \text{where } b_i \in \{0,1\} \text{ and } b_i = 1 \text{ iff } i \in s$$

For large sup, however, this function is impractical since it takes huge values of order $O(2^{\sup})$. Further, the obtained constraint propagation is not satisfactory.

In case the cardinality of all relevant sets is fixed, say for some s to k such that $s = \{n_1, \ldots, n_k\}$, one can do much better by ordering the integers n_1 through n_k and interpreting them as a number to the base $\sup+1$.[3]

$$(n_1, \ldots, n_k)_{\sup+1} \tag{13.1}$$

Having references N_1 through N_k to the elements n_1 through n_k one can state the fact that they must be ordered through the finite domain integer propagators

$$N_1 \texttt{<:}\ N_2 \texttt{<:}\ \ldots \texttt{<:}\ N_k\ . \tag{13.2}$$

This gives strong constraint propagation whenever the bounds of some N_i are narrowed. The library procedure `{FS.int.match S DV}` supports the rank function (13.2) more immediately.

$$\begin{aligned} \texttt{\{FS.int.match S DV\}} \equiv \quad & S = \{I_1, \ldots, I_n\} \wedge DV = \langle I_1, \ldots, I_n\rangle \wedge |S| = n \\ \wedge \quad & I_1 \in S \wedge \ldots \wedge I_n \in S \wedge I_1 < I_2 \wedge \ldots \wedge I_{n-1} < I_n \end{aligned}$$

Informally, `FS.int.match` constrains the elements of the list `DV` to the n elements of `S` and vice versa. The propagator for the *rank*-function for subsets of $\{1, \ldots, n\}$ with (uniformly) fixed cardinality 3 can now be implemented as follows.

[3] Note that this does not require all sets to have the *same* fixed cardinality!

```
fun {Rank N S} X1 X2 X3 R  in
   {ForAll [X1 X2 X3 R] FD.decl}
   {FS.int.match S [X1 X2 X3]}
   {FD.sumC [N*N N 1] [X1 X2 X3] '=:' R} R
end
```

Function `Rank` first defines its local variables and constrains them to finite domains. Then, the variables `X1`, `X2` and `X3` are matched against the three individual elements of set `S`. Finally, `R` is constrained to the rank $R = N^2 \times X1 + N \times X2 + X3$ and returned.

The effect of this rank function is examined by reconsidering the Steiner problem. The procedure `BreakSymmetries` imposes an ascending order on the individual sets. Note that this has to be compatible with the distribution strategy. This is the case for naïve distribution since it starts with the smallest element of the first set and carries on with the next bigger element at the next set.

```
proc {BreakSymmetries N Ss} Rs in
   Rs = {Map Ss fun {$ S} {Rank N S} end}
   {ForAllTail Rs
    proc {$ T} case T of R1|R2|_ then R1 <: R2 else skip end
    end}
end
```

First, a list `Rs` of all ranks is constructed by mapping the individual set `S` of `Ss` to their corresponding ranks via `Rank`. Then, the order on the sets is established by imposing the `<:`-constraint on all two adjacent corresponding ranks `R1` and `R2` using `ForAllTail` and a pattern matching **`case`**-statement.

The script for the Steiner problem can now be improved by applying `BreakSymmetries`:

```
fun {Steiner N}
   proc {$ Ss}
      {SteinerConstraints N Ss}
      {BreakSymmetries N Ss}
      {FS.distribute naive Ss}
   end
end
```

These (logically) redundant ordering constraints significantly reduce memory consumption and runtime for this problem: The speed-up factor for the Steiner problem of order 9 is 6.3 and memory consumption reduces by a factor of 5.9. Table 13.1 shows the number of choice points and failures of formulations of the Steiner problem with and without ordering constraints. The columns labelled *steiner(n)** refer to formulations using the redundant ordering constraints while columns labelled *steiner(n)* refer to the implementations using no ordering constraints.

13.2.2 Scheduling a Golf Tournament

This section discusses a program that finds a schedule for a golf tournament. The problem was initially posted on the newsgroup `comp.constraints` and `sci.`

problem	steiner(7)	steiner(7)*	steiner(9)	steiner(9)*
choice points	20	15	4545	565
failures	6	1	4521	541

Table 13.1: Comparing the number of choice points and failures for formulations of the Steiner problem with and without ordering constraint.

`op-research` by Harvey. Suppose there are 32 golfers who play in groups of 4, so-called *foursomes*. For every week of the golf tournament new sets of foursomes are to be compiled. The task is to assign foursomes for a maximum number of weeks such that no golfer plays with another golfer in a foursome twice.

The Upper Bound for Number of Weeks The upper bound for the number of weeks is 10 weeks due to the following argument: There are $\binom{32}{2} = 496$ pairing of players. Each foursome takes 6 pairings and every week consists of 8 foursomes, hence, every week occupies 48 pairings. Having only 496 pairings available, at most $\lfloor 496/48 \rfloor = 10$ weeks can be assigned without duplicating foursomes.

Modeling the Problem A foursome is modeled as a set of cardinality 4. A week is a collection of foursomes and all foursomes of a week are pairwise disjoint and their union is the set of all golfers. This leads to a partition constraint. Furthermore, each foursome shares at most one element with any other foursome because a golfer must never meet another golfer twice in a foursome. Therefore, the cardinality of the intersection of a foursome with any other foursome of the other weeks has to be either 0 or 1.

The distribution strategy is crucial for this problem. It was proposed by Novello: A player is taken and assigned to all possible foursomes. Then the next player is taken and assigned and so on. This player-wise distribution makes it possible to solve instances of the golf problem for up to 9 weeks. The approach which is usually coming to mind first, namely to explore all possible foursomes of a variable before moving on to the next variable, fails even for smaller instances of the problem.

The Propagation Algorithm The procedure `GolfSolver` implements the propagation algorithm. It obtains the number of weeks (`NbWeeks`), the number of foursomes per week (`NbFourSomes`) and the number of players (`NbPlayers`), and returns the list of weeks of length `NbWeeks`. Every week is represented by a list of `NbFoursomes` foursomes. An individual foursome is a set with cardinality 4 and a subset of $\{1, \ldots, \texttt{NbPlayers}\}$. Each golfer is identified by an individual integer $i \in \{1, \ldots, \texttt{NbPlayers}\}$.

```
proc {GolfSolver NbWeeks NbFourSomes NbPlayers Weeks}
   Weeks = {MakeList NbWeeks}
   {ForAll Weeks
    proc {$ Week}
       Week = {MakeList NbFourSomes}
       {ForAll Week proc {$ FourSome}
                       {FS.var.upperBound 1#NbPlayers FourSome}
                       {FS.cardRange 4 4 FourSome}
```

```
                    end}
         {FS.partition Week {FS.value.make 1#NbPlayers}}
      end}
```

Procedure `GolfSolver` creates first a list `Weeks` holding `NbOfWeeks` weeks and each element of `Weeks` is constrained to a list of `NbOfFourSomes` foursomes. Every foursome `FourSome` is constrained to $\emptyset \subseteq$ `FourSome` $\subseteq \{1, \ldots,$ `NbOfPlayers`$\} \wedge$ $|$`FourSome`$| = 4$ by the finite set library abstractions `FS.var.upperBound` and `FS.cardRange`, respectively. All foursomes of a week partition the set of all players $\{1, \ldots,$ `NbOfPlayers`$\}$.

The following nested `ForAllTail`- and `ForAll`-loops impose that every foursome shares at most one element with any other foursome of other weeks.

```
    {ForAllTail Weeks
     proc {$ WTails}
        case WTails
        of Week|FolWeeks then
           {ForAll Week
            proc {$ FourSome}
               {ForAll FolWeeks
                proc {$ FourSomesOfWeek}
                   {ForAll FourSomesOfWeek
                    proc {$ FourSomeOfWeek}
                       {FS.cardRange 0 1
                        {FS.intersect FourSome FourSomeOfWeek}}
                    end}
                end}
            end}
        else skip end
     end}
end
```

The outermost `ForAllTail`-loop in conjunction with the **`case`**-statement iterates over the weeks and extracts the current week `Week` and the following weeks `FolWeeks`. The `ForAll`-loops iterate over the foursomes in `Week` and enforce that all foursomes in `Week` do not share more than one player with the foursomes in `FolWeeks`.

The Distribution Algorithm The procedure `DistrPlayers` implements the player-wise distribution strategy (round-robin).

```
proc {DistrPlayers NbPlayers Weeks}
   {For 1 NbPlayers 1
    proc {$ I}
       {ForAll Weeks
        proc {$ AllFoursomes}
           {ForAll AllFoursomes
            proc {$ FourSome}
               {FS.distribute generic(element: fun {$ _} I end
                                      rrobin: true)
```

```
                [FourSome]}
            end}
        end}
    end}
end
```

The outer-most `For`-loop[4] iterates over all players while the `ForAll`-loops iterate over the individual foursomes. The inner-most loop calls the finite set library abstraction `FS.distribute` for a single foursome `Foursome`. The distribution strategy is instructed to branch over `I` (`element: fun {$ _} I end`) and to proceed to the next variable after every distribution step (`rrobin: true`).

Combining Propagation and Distribution to a Solver The solver is obtained by combining propagation and distribution algorithms in the function `Golf` which returns an instance of a solver for a given number of weeks (`NbWeeks`) and foursomes (`NbFourSomes`).

```
fun {Golf NbWeeks NbFourSomes}
   proc {$ Weeks} NbPlayers = 4*NbFourSomes in
      {GolfSolver NbWeeks NbFourSomes NbPlayers Weeks}
      {DistrPlayers NbPlayers Weeks}
   end
end
```

To find a solution for 9 weeks and 8 foursomes in Mozart Oz, simply invoke the solver by `{SearchOne {Golf 9 8}}`.

Discussion Walser conducted tests with CPLEX [70] but running his program for hours produced no solution for more than 8 weeks and failed to find a solution for 9 weeks [147]. In Mozart Oz, the program discussed in this section finds a solution for 9 weeks and 8 foursomes in a couple of seconds producing a search tree with 215 choice points and a depth of 200. To the authors' best knowledge, no solutions for more than 9 weeks have been found by now and solutions for 9 weeks were only found by set constraint solvers.

Sets with Attributed Elements

Many practical problems require an association of set elements with some attributes. In the golf scheduling example every foursome may be required to contain an instructor to make sure that every golfer in a foursome behaves appropriately. The list `InstructorList` defines for every player whether he or she is an instructor or not.

```
InstructorList = [0 1 0 0 1 0 1 0 0 0 1 0 0 1 0 1
                  0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 1]
```

A golfer represented by an integer i is identified to be an instructor if the i-th element of `InstructorList` is 1. Otherwise, *i.e.* the i-th golfer is no instructor, the i-th element is 0.

[4] `{For 1` n `1 P}` $\equiv$ `{P 1} {P 2} ... {P` $n-1$`} {P` n`}`

Procedure `Instructors` enforces that every foursome contains at least one golf instructor.

```
proc {Instructors NbPlayers Weeks InstructorList}
   {ForAll Weeks
    proc {$ AllFoursomes}
       {ForAll AllFoursomes
        proc {$ FourSome} BL in
           {FS.reified.areIn 1#NbPlayers  FourSome BL}
           {FD.sumC InstructorList BL '>=:' 1}
        end}
    end}
end
```

The nested `ForAll`-loops iterate over every foursome `FourSome`. A golfer i in a foursome is attributed with the information whether or not he or she is an instructor by reification. That means, the membership of i is reflected to a $0/1$-variable b_i: $(i \in \texttt{FourSome} \rightarrow b_i = 1) \vee (i \notin \texttt{FourSome} \rightarrow b_i = 0)$. To enforce the at least one instructor is in each foursome, this variable is multiplied with the corresponding i-th element in the `InstructorList` and the sum of the products must be equal or greater than 1:

$$\sum_{i=1}^{\texttt{NbPlayers}} \texttt{InstructorList}_i \times \texttt{BL}_i \geq 1$$

The library procedure call `{FS.reified.areIn 1#NbPlayers FourSome BL}` realizes this constraint: for all $i \in \{1, \ldots, \texttt{NbPlayers}\}$ the membership $i \in \texttt{FourSome}$ is reflected in the corresponding $0/1$-variable in the list `BL`. The in-equation is implemented by the finite domain propagator `{FD.sumC` a_i x_i `'>=:'` c`}` which implements the constraint $\sum_i a_i \times x_i \geq c$.

Extending the Solver The solver for the golf scheduling problem can be extended to a solver taking the instructor constraints into account by simply calling the procedure `Instructors`:

```
fun {GolfInstructor NbWeeks NbFourSomes}
   proc {$ Weeks} NbPlayers = 4*NbFourSomes in
      {GolfSolver NbWeeks NbFourSomes NbPlayers Weeks}
      {Instructors NbPlayers  Weeks InstructorList}
      {DistrPlayers NbPlayers Weeks}
   end
end
```

13.2.3 Dependency Parsing

This section illustrates the benefits of finite set constraint programming and in particular the unique features of Mozart's finite integer constraint library in the area of computer linguistics. To this end, some key ideas described by Duchier in [44] for building a

dependency parser are reproduced here. The dependency parser finds possible readings of a phrase in a natural language.

Duchier proposes to use mutually dependent trees to obtain these readings: a syntax tree (ID tree) and a topological tree per reading (LP tree). While a syntax tree does not reflect any ordering on words, a topological tree represents a possible reading of the phrase by imposing an ordering on the words of the phrase.

Suppose the following German phrase (taken from [44]): "(dass) Maria einen Mann wird lieben können". This phrase has various readings:

1. "(dass) Maria einen Mann lieben können wird"
2. "(dass) Maria einen Mann wird lieben können"
3. "(dass) Maria wird einen Mann lieben können"

Figure 13.1 shows the ID tree for the example phrase.

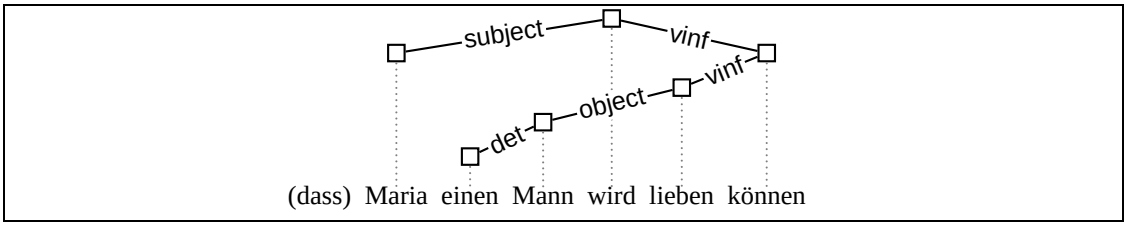


Figure 13.1: ID tree.

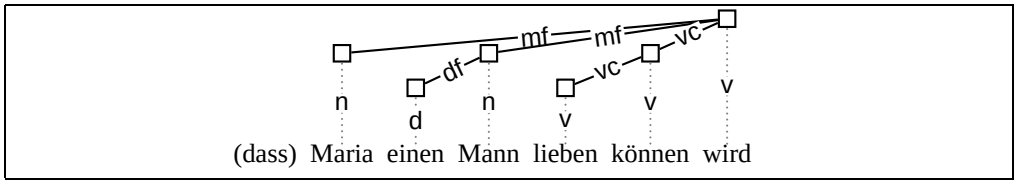
LP trees represent possible linear word ordering of this phrase as shown in Figure 13.2. In an LP tree, the edges and nodes are labelled.

Thereby, an LP tree has to obey a configuration in conjunction with an order on the labels of the tree. A configuration is defined by a lexicon and determines the number of outgoing edges per node and the labels of the edges. A label of an edges is called external label taken from $\mathcal{L}_{\text{LP}} = \{\mathsf{df}, \mathsf{mf}, \mathsf{vc}, \mathsf{xf}\}$ while a label of a node is called internal label taken from $\mathcal{L}_{\text{INT}} = \{\mathsf{d}, \mathsf{n}, \mathsf{v}\}$ (dotted vertical lines in Figure 13.2). The disjoint union of the labels is totally ordered and in the example: $\mathcal{L}_{\text{LP}} \uplus \mathcal{L}_{\text{INT}} : \mathsf{df} < \mathsf{d} < \mathsf{n} < \mathsf{mf} < \mathsf{vc} < \mathsf{v} < \mathsf{xf}$. This order induces a left-to-right precedence on a given node and its the daughter nodes. This precedence is partial since daughters with same labels can be freely permuted. As example, consider the top-most node of Figure 13.2(c) which obeys the total order: $\mathsf{mf} < \mathsf{v} < \mathsf{xf}$. Moving on to the right-most node, the obeying order is: $\mathsf{mf} < \mathsf{vc} < \mathsf{v}$. The total order on the labels is met by all nodes and their outgoing edges.

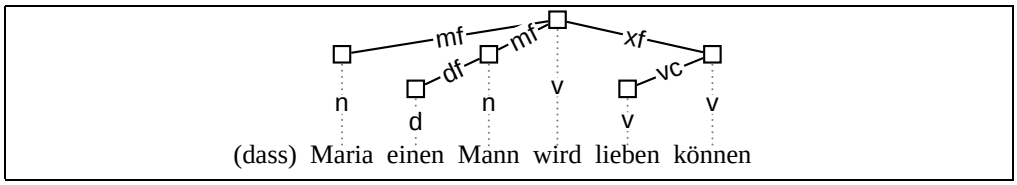
This section discusses (i) how these trees can be represented by finite integer set constraints and (ii) how they can be made ordered and projective to represent a valid word ordering for a certain reading. A formal discussion of the described scheme is given by Duchier in [44].

Labeled Trees

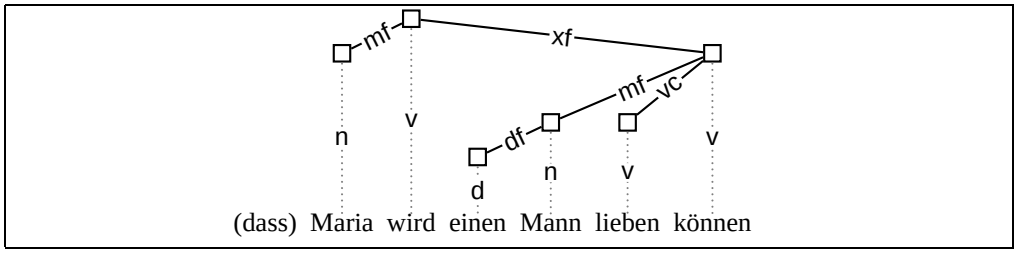
ID trees and LP trees are labeled trees. T(V, $\mathcal{L}$) denotes the set of finite trees (V, E) with nodes V, labeled edges $E \subseteq \text{V} \times \text{V} \times \mathcal{L}$ and set of labels $\mathcal{L}$. A node represents a word



(a) 1st reading .



(b) 2nd reading .



(c) 3rd reading .

Figure 13.2: LP trees corresponding to readings of enumerated list on page 136.

w of a phrase and is identified with an integer. For every label $\ell \in \mathcal{L}$ there is a function $\ell(w) = \{w' \mid (w, w', \ell) \in \mathrm{E}\}$ denoting a set of words reachable directly by ℓ-edges from w. Function $\ell(w)$ is the base for the functions stating the well-formedness condition of a labelled tree shown in Figure 13.3. This condition has to met by a graph (V, E) to belong to T(V, $\mathcal{L}$). Representing the ℓs and ws with integers leads to a finite integer set CSP.

$$
\begin{aligned}
&\psi(V, \mathcal{L}) \;\equiv\; \\
&\qquad \mathsf{root} \subseteq V \;\wedge\; |\mathsf{root}| = 1 \\
&\wedge \quad V = \mathsf{root} \;\uplus\; \uplus\{\mathsf{daughters}(w) \mid w \in V\} \\
&\wedge\, \forall w \in V \\
&\qquad\quad \mathsf{eqdown}(w) = \{w\} \uplus \mathsf{down}(w) \\
&\wedge \qquad \mathsf{down}(w) = \cup\{\mathsf{eqdown}(w') \mid w' \in \mathsf{daughters}(w)\} \\
&\wedge \qquad \mathsf{equp}(w) = \{w\} \uplus \mathsf{up}(w) \\
&\wedge \qquad\quad \mathsf{up}(w) = \cup\{\mathsf{equp}(w') \mid w' \in \mathsf{mother}(w)\} \\
&\wedge \quad \mathsf{daughters}(w) = \uplus\{\ell(w) \mid l \in \mathcal{L}\} \\
&\wedge \quad \mathsf{mother}(w) \subseteq V \;\wedge\; |\mathsf{mother}(w)| \leq 1 \\
&\wedge \quad \forall w' \in V \quad w' \in \mathsf{daughters}(w) \;\equiv\; w \in \mathsf{mother}(w')
\end{aligned}
$$

Figure 13.3: Well-formedness condition of labeled trees.

Imposing Linear Precedence

An LP-tree is required to be *projective* and *well-ordered* to obtain a continuous ordering on the words of a phrase. This is achieved by linear precedence trees (V, E, I, $\prec$) where (V, E) is a labelled tree in T(V, $\mathcal{L}$), I : V $\rightarrow \mathcal{L}_{\text{INT}}$ is a function mapping nodes to internal labels, and $\prec$ is a total order on V.

Projective An LP-tree is projective if

$$\forall w \in V \quad \mathsf{eqdown}(w) \text{ is } \prec\text{-convex in } V \tag{13.3}$$

where a set S is $\prec$-convex in V iff $\forall x, y \in S \subseteq V,\ \forall z \in V\ x \prec z \prec y \Rightarrow z \in S$ relative to a total order $\prec$ on V. The intuition behind this condition is to have no gap in the orderings denoted by any subtree of the LP tree. Since words are identified with integers, this condition is directly provided by the set library propagator `FS.int.convex`.

Well-ordered. The total order $\prec$ on V is defined by:

$$\forall S_1, S_2 \subseteq V:\ S_1 \prec S_2 \ \equiv\ \forall w_1 \in S_1, \forall w_2 \in S_2\ w_1 \prec w_2.$$

A tree is well-ordered if whenever $\ell_1 < \ell_2$ the downset at label ℓ_1 precedes the downset at label ℓ_2 according to the order $\prec$. The down set $\mathsf{down}(w)^\ell$ for an internal label ℓ is $\{w\}$ if $\mathrm{I}(w) = \ell$ (and $\emptyset$ otherwise) while for an external label ℓ, the down set is the set of all words of the subtree connected by the ℓ-edge, *i.e.*, $\mathsf{down}^\ell(w) = \cup\{\mathsf{eqdown}(w') \mid w' \in \ell(w)\}$. The down sets have to meet an well-ordering condition according to the order on the labels: $\mathcal{L}_{\text{LP}} \uplus \mathcal{L}_{\text{INT}} = \{\ell_1, \ldots, \ell_n\} : \ell_1 < \ldots < \ell_n$:

$$\mathsf{down}^{\ell_1}(w) \prec \cdots \prec \mathsf{down}^{\ell_n}(w) \tag{13.4}$$

The order condition (13.4) can be directly encoded by the `FS.int.seq`-propagator because of the integer encoding of individual words. The well-formedness condition in conjunction with conditions (13.3) and (13.4) form again a CSP over finite sets of integers.

Discussion This section discusses a part of a constraint model for a dependency parser to emphasize the benefits of using finite integer set constraints for an implementation of such a parser. The complete parser has been conveniently implemented in Mozart Oz. The main advantage of using Mozart is the availability of set propagators taking the integer domain into account, as `FS.int.seq` and `FS.int.convex`. As reported in [44], the implemented parser finds in many cases different readings without any failed nodes in the search tree.

13.3 Performance Evaluation

This section compares the performance of the finite integer set constraint solver of Mozart Oz to the finite integer set solvers of ILOG SOLVER 5.0 and *ECL^{i}PS^e* 5.2. The measurements are conducted on the same platform as in Section 10.2, *i.e.*, Linux with

kernel 2.2.16-22, 256MB, Athlon 700MHz. In contrast to Section 10.2, the efficiency of solving application programs including search is compared rather than measuring the plain propagation performance. First, the efficiency of solving instances of the Steiner problem (Section 13.2.1) and the golf tournament scheduling problem (Section 13.2.2) with Mozart Oz 1.2.0 is compared with the corresponding solvers of ILOG SOLVER 5.0 and *ECL^{i}PS^e* 5.2. Second, alternative Mozart solvers for the Steiner problem and the golf scheduling problem are compared with Mozart solvers based on finite integer set constraints. The programs used for benchmarking can be found at [105].

Benchmarking against *ECL^{i}PS^e* and ILOG SOLVER The finite set solver of Mozart is compared with the solver `fd_sets` of *ECL^{i}PS^e* and the set solver of ILOG SOLVER. An instance of the Steiner problem for $n = 9$ and no symmetry breaking is used. The used instance of the golf scheduling problem employs a naïve distribution strategy and finds a solution for 5 weeks and 8 foursome per week. The benchmark programs were run ten times to obtain stable results. Table 13.2 shows the speed-ups against the corresponding Mozart solvers. A speed-up "factor" annotated with "factor^{-1}" means the corresponding solver is better than Mozart. Otherwise Mozart is faster. The re-computation depth of the Mozart search engines is 1. The figures in the row "steiner*" were generated with the Steiner formulation using the symmetry breaking constraint.

problem	*ECL^{i}PS^e* **5.2**	**ILOG SOLVER 5.0**
steiner	3.74	4.73^{-1}
steiner*	16.66	1.06^{-1}
golf	3.58	2.29^{-1}

Table 13.2: Speed-ups against ILOG SOLVER 5.0 and *ECL^{i}PS^e* 5.2 .

The `fd_sets`-solvers of *ECL^{i}PS^e* are constantly slower than the corresponding Mozart solvers for Steiner and golf solvers. This is not surprising when considering the results on plain propagation performance discussed in Section 10.2.2. ILOG SOLVER performs better on the given problems especially for the golf scheduling problem. The Steiner solver performs almost no propagation (it checks the cardinality of intersections) while the golf solver can do much better due to the partitioning constraints. A possible reason for this behavior is that Mozart uses copy-based search which performs poorly on solvers with little propagation and goes well for strongly propagating solvers. This is because a whole computation space is copied per distribution step no matter of how many variables have been changed. Further, a re-computation depth of 1 requires the most copying and tuning this parameter could certainly improve the performance. The figures for the Steiner formulation with the symmetry breaking constraint (row "steiner*") show that this deficiency can be cured by sophisticated propagation algorithms taking advantage of the domain of integers. These constraints are unique to Mozart.

Although ILOG SOLVER runs faster than the finite integer set solver of Mozart, the Mozart solver has proved to be competitive with available state-of-the-art solvers.

Comparing Different Set Solvers in Mozart Solvers based on the finite integer set

library are compared with solvers based on characteristic functions of sets. A characteristic function is a vector of 0/1-variables for every possible element of a set. The implementation is straightforward with finite domain constraints but for sets with larger cardinalities inefficient *w.r.t.* memory consumption. Characteristic function solvers for the Steiner problem and the golf scheduling problem are implemented for comparison.

Table 13.3 shows the speed-ups and the heap factor obtained by comparing the different solvers. A heap factor denotes the factor of memory used to solve a problem compared to the implementation using the finite set library.

problem	**speed-up**	**heap factor**
steiner (characteristic function)	1.45	1.51
golf (characteristic function)	15.6	8.55

Table 13.3: Comparing run-time (speed-up) and heap memory consumption (heap factor) of various solvers for the steiner and the golf problem.

While the characteristic function solver for Steiner is equally efficient as the finite set solver, due to only ternary sets in the Steiner problem, the solver for the golf problem makes the deficiencies of the characteristic function implementation obvious. More than 8 times the memory is needed slowing down the whole solver by a factor of about 16. Hence, straightforward implementations of solvers based on characteristic function are not suitable for general finite set solver. For certain problems, however, solvers based on characteristic function may be worth to be considered.

13.4 Related Work

Finite Sets

Available finite set constraint solvers are solvers over finite sets of integers as the set solver of ILOG SOLVER [73, 115] and the `fd_sets` solver of *ECL^{i}PS^e* [76]. The `fd_sets` solver is the successor of Gervet's finite set constraint solver over Herbrand terms CONJUNTO [5] [54] which was the first available solver. This situation supports the claim that finite integer sets provide the expressiveness required by todays practical constraint problems.

Finite set constraints have been successfully used for applications in computer linguistics [48, 80, 45] as demonstrated in Section 13.2.3. Mozart's set constraint solver is particularly well-suited for these applications since only this solver provides special propagators (namely `FS.int.convex` and `FS.int.seq`) designed for computer linguistic applications.

For applications which naturally come in mind, as plain set partitioning, set constraints cannot compete with integer linear programming techniques [65, 66] for large

[5] This solver handles even power sets of Herbrand terms.

instances of such problems. A case study about the achievable performance of finite integer set constraint solving for set partitioning problems [100] confirmed that. Although a sophisticated pre-processing technique and tailored filters have been proposed, the size of the instances is orders of magnitudes smaller than for linear integer programming techniques.

Other Approaches

More complex systems provide for a regular set description language. This includes the work by Walinsky on CLP(Σ^*)[146] which deals with regular sets of *words*, as well as Foster's CLP(SC) [49] (proposed by Kozen [81]), which deals with regular sets of *trees*. Regular sets of trees have been particularly prominent in static program analysis (see [113] for overviews and references) and several specialised solvers have been developed. In this domain, constraint solving usually means testing satisfiability of a constraint, or emptiness of a set variable in all solutions (or a distinguished solution) of a constraint.

A third approach allows set descriptions of the form $\{X, Y\}$ (also called *set terms*) where X and Y are variables denoting elements, and provide an associative, commutative, and idempotent unification procedure. This is the approach of systems like CLPS [84], {log} [43], CLP($\mathcal{SET}$) [42], and others [9, 82, 139]. Yet different approaches allow set comprehensions like $\{x \mid p(x)\}$ with an intensional semantics [19], or consider non-standard set domains for interpretation of cyclic set descriptions like $X = \{X, \{X\}\}$ [86].

Part III

First-class Constraints

Chapter 14

Promoting Constraints to First-class Citizens

This chapter promotes non-basic constraints to first-class citizens, short first-class constraints. After presenting the idea, an abstract data type for first-class constraints is defined and syntactic support for them in the language Oz is proposed. Then, various programming techniques for first-class constraints are presented which demonstrate the benefits with the help of case studies. These case studies have been programmed using a prototype implementation of first-class constraints for Mozart. The integration of first-class constraints into an existing solver is discussed next. Finally, this chapter closes with discussing related work.

14.1 The Idea

Traditionally, constraint propagation of non-basic constraints reasons about the values their parameters can take, called *domain propagation*. This can be limited through the restricted view of the individual non-basic constraints.

Promoting constraints to first-class citizens overcomes this deficiency by adding an extra level of propagation to the constraint solver, so-called *symbolic propagation*. Constraints being first-class citizens are called *first-class constraints*.

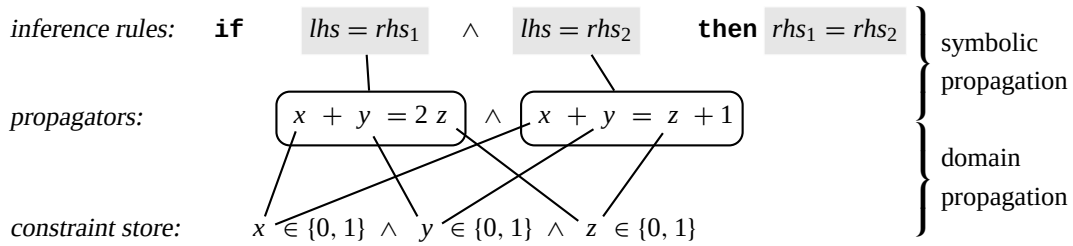


Figure 14.1: Idea of first-class constraints: combining domain propagation and symbolic propagation to hybrid propagation.

Figure 14.1 demonstrates the use of first-class constraints. Regard the constraints:

$$x + y = 2z \ \wedge \ x + y = z + 1 \ \wedge \ x \in \{0, 1\} \ \wedge \ y \in \{0, 1\} \ \wedge \ z \in \{0, 1\}.$$

Traditional domain propagation leaves the domains of x, y, and z untouched. Performing symbolic propagation, the right hand-sides of the constraints $x + y = 2z$ and $x + y = z + 1$ are equated to a new constraint $2z = z + 1$. Domain propagation of $2z = z + 1$ determines z to 1 and eventually, x to 1 and y to 1. Results of symbolic propagation are immediately visible for domain propagation and vice verse. This combination of symbolic propagation and domain propagation is called *hybrid propagation*.

The availability of symbolic propagation makes new propagation techniques possible which are developed in the referred sections.

Early Failure Detection (Section 14.3.1) Due to the limited view of a single constraint on the constraint store, reasoning and especially failure detection is limited. Often recognizing a certain constraint pattern makes it possible to spot an inconsistency much earlier than constraint propagation can do and sometimes constraint propagation on its own is not able to detect the inconsistency at all. For example $x < y \wedge y < x$ is obviously inconsistent. But the time ordinary finite domain propagation takes to detect the inconsistency is proportional to the domain size of x and y, and hence, can be quite long. Reasoning about the constraints themselves can detect the unsatisfiability of this constraint immediately.

Constraint Optimization (Section 14.3.2) Constraints fed into a constraint solver can often be improved regarding their propagation behavior. Common sub-constraints, for example, can be collapsed and constraints can be reformulated to provided for better domain pruning.

Garbage Collection (Section 14.3.3) Usually constraints are garbage collected as soon as they are entailed by the constraint store. But typically that requires the parameter of the constraints to be determined. In many cases constraints could be discarded earlier. Consider the finite domain constraint $x + 1 = z \wedge x \leq z$. The constraint $x \leq z$ can be discarded since it is implied by $x + 1 = z$.

Smallest Sets of Inconsistent Constraints (Section 14.3.4) Like every kind of programming, constraint programming is prone to error. A common programming error is to put up an incorrect model a given problem or to implement a constraint model incorrectly. This frequently results in inconsistent constraints which cause immediate failure. Debugging such symptoms is supported by finding sets of constraints that are responsible for the inconsistency.

14.2 Constraints as Values

This section introduces first-class constraints as values of an abstract data type. A first-class constraint is implemented by a *first-class propagator*.

First-Class Propagators A first-class propagator is a value of an abstract data type and is hence defined in terms of its operations. It can be handled like any other primitive

value, i.e, it can be part of composite data structures or can be used in applications or expressions.

Creation First-class propagators are created by imposing propagators using

```
prop [inactive] {Cs} σ end
```

where `Cs` is constrained to a list of first-class propagators referring to the propagators imposed by σ on the current thread. If σ does not impose any propagator on the current thread then `Cs` is `nil`. In case **`inactive`** is added, the propagators are imposed with propagation turned off. Otherwise propagation is turned on by default.

Abstract Data Type The abstract data type for first-class propagators provides the following abstractions:

`Constraint.is`: *Propagator* → *Boolean*
: Checks a value to be a propagator.

`Constraint.activate`: *Propagator*
: Activates propagation of a propagator.

`Constraint.deactivate`: *Propagator*
: Deactivates propagation of a propagator.

`Constraint.discard`: *Propagator*
: Discards a propagator.

`Constraint.isEntailed`: *Propagator* → *Boolean*
: Checks a propagator for entailment.

`Constraint.getName`: *Propagator* → *Atom*
: Obtains the name of a propagator.

`Constraint.getParameters`: *Propagator* → *List of Value*
: Obtains the parameters of a propagator.

`Constraint.getKey`: *Propagator* → *Atom*
: Obtain key for a propagator.

`Constraint.identifyParameters`: *List of Value* → *List of Integer*
: Identifies aliased parameters.

`Constraint.reflectSpace`: *List of Value* → *List of Value* × *List of Propagator*
: Reflects all variables and propagators of a space reachable from a given list of variables.

Note that reflective operations are typically *non-monotonic*, *i.e.*, the produced result depends on the current state of the solver. Hence, these operations can be safely applied only if propagation has reached a fixed-point. This has to be taken into account when adding new basic constraints to the constraint store while reasoning over first-class constraints. Adding new basic constraints typically requires the re-computation of the fixed-point resulting in a changed set of first-class constraints to reason about.

Type Test A variable can be checked to refer to a propagator by

```
{Constraint.is C B}
```

where `B` is bound to **`true`** if `C` refers to a propagator and to **`false`** otherwise.

Controlling Propagation Propagation of a propagator referred to by `C` can be turned on by `{Constraint.activate C}` while `{Constraint.deactivate C}` turns propagation off.

Entailment Programming with first-class propagators typically involves rewriting sets of propagators to more efficient formulations. That requires discarding the redundant propagator which is replaced. Furthermore, reasoning about propagators may take into account that a propagator has already become entailed by the constraint store, *i.e.*, can be ignored. A propagator referred to by `C` is explicitly entailed by `{Constraint.discard C}`, *i.e.*, `C` is discarded from the computation space. By discarding a propagator, its whole host space may become entailed. A propagator `C` can be checked to be entailed by `{Constraint.isEntailed C B}` which binds `B` to **`true`** if `C` is entailed, no matter whether explicitly by `discard` or by entailment through the constraint store. If `C` is not yet entailed `B` is bound to **`false`**.

Reflection In the course of symbolic propagation, propagators and their parameters have to be identified and reflected to primitive values.

The name of a propagator referred to by `C` can be obtained by

```
{Constraint.getName C N}
```

which binds `N` to an atom representing the name of `C`. The parameters of a propagator can be retrieved by `{Constraint.getParameters C Ps}` which binds `Ps` to the parameters of `C`.

Additionally, the proposed operations are useful and convenient in the applications discussed in Section 14.3. A key of a propagator can be obtained by `{Constraint.getKey C K}` which binds `K` to an atom representing a key for `C`. Such a key makes efficient storage and retrieval of first-class propagators (Section 14.3.2) by dictionaries possible because propagators with the same name identically imposed on the same variables produce the same key.

It is often convenient to obtain all propagators of the current computation space at once. That can be done by

```
{Constraint.reflectSpace Rs Vs Cs}
```

which takes a list `Rs` of variables. It collects all propagators of the current space that have at least one undetermined variable of `Rs` as a parameter. Furthermore, `reflectSpace` collects propagators which share undetermined parameters with collected propagators. Thus, the transitive closure of all propagators "reachable" from `Rs` is computed. The collected propagators are returned as list `Cs`. Additionally, `Vs` is bound to the list of all undetermined variables occurring as parameters of the propagators in `Cs`. Reflecting a whole space makes it possible to use first-class propagators in an orthogonal way since the original constraint program needs not be modified to obtain first-class propagators referring to the propagators in the current computation space.

Synchronization The operations `activate`, `deactivate`, `discard`, `isEntailed`, `getName`, `getParameters`, and `getKey` synchronize on their parameter `C` to be bound to a first-class propagator.

Identification The ==-operator (Figure 3.1 on page 16) has been extended to identify first-class propagators to refer to identical propagators.

Variables can be identified by

```
{Constraint.identifyParameters Vs Ids}
```

which maps the list of variables `Vs` to a list of integer identifiers `Ids` by assigning to each element in `Vs` the index of its first occurrence in `Vs`. Thus aliased variables can be detected easily. (Note that as discussed in Section 9.3, the same idea is used by the CPI to detect aliased variables in vectors of parameters.)

14.3 Programming

This section discusses several programming techniques for first-class constraints and demonstrates their benefits by various case studies. These presented techniques perform either some reasoning about a set of constraints in the same fashion as inference rules or take advantage of the possibility to turn propagation of individual propagators on and off. The programs presented in this section can be found at [105].

14.3.1 Early Failure Detection

One of the major goals of constraint programming is to avoid exploration of parts of the search tree that do not contain any solutions. But there are cases where propagation takes significant time to detect failure or is even unable to do so. An example for potential long lasting propagation are the finite domain constraints $x < y \wedge y < x$ and $2x = y \wedge 2u = v \wedge y + 1 = v$ assuming sufficiently large domains. An example for an unsatisfiable constraint that cannot be spotted without any meta reasoning is $x, y, z \in \{0, 1\} \wedge x \neq y \wedge x \neq z \wedge y \neq z$.

This section demonstrates how meta constraint programming can be used to detect unsatisfiable constraints where ordinary constraint propagation fails to do so. Thus the search tree can be significantly pruned and bigger instances of the problem can be solved.

A modified Hamiltonian path problem is used as example, where the aim is to find a path through a given directed graph from an arbitrary starting node to an arbitrary ending node such that all nodes of the graph are visited once and the path is valid for the reverse direction too.

The Constraint Model and its Implementation The problem data is given as set *Arcs* of 2-tuples $arc(f, T)$, where the set $T \subseteq \{1, \ldots, n\}$ contains all nodes $t \in T$ such that there is an arc from node f to t. Every of the n nodes of the graph is represented by a finite domain variable $x_i \in \{1, \ldots, n\}$ which represents the position of the ith node; the variables have to be pairwise distinct (constraint (14.1)). Constraint (14.2) expresses the path from the starting node to the ending node. Node x_i is the successor of x_f if $x_i = x_f + 1$ holds. Note the extra clause for the ending point. The constraint (14.3) is dual to constraint (14.2) and models the reverse path.

$$distinct(x_1, \ldots, x_n) \tag{14.1}$$

$$\forall\ arc(f, T) \in Arcs : \bigvee_{i \in T} (x_i = x_f + 1) \vee x_f = n \tag{14.2}$$

$$\forall\ arc(f, T) \in Arcs : \bigvee_{i \in T} (x_i + 1 = x_f) \vee x_f = 1 \tag{14.3}$$

The constraint model is implemented one-to-one with Mozart finite domain constraints and uses disjunctive combinators producing choice-points to obtain the same behavior as the program used in [61]. The search strategy is naïve, *i.e.*, it picks from the left-most finite domain variable x_l the minimum element m and creates a choice-point $x_l = m \vee x_l \neq m$.

Deriving an Early-Failure Criterion Deriving a criterion is a creative process and it is hard to give any guidelines. But it is helpful to have a tool handy that displays the constraints in a node of the search tree. The Propagator Viewer[1] in combination with the Oz Explorer offers this functionality.

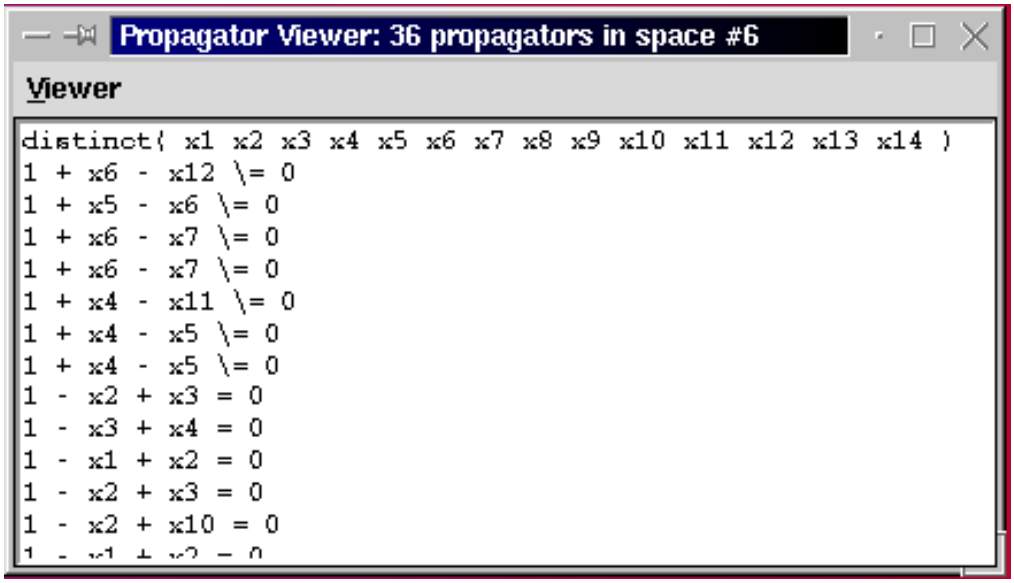


Figure 14.2: The Propagator Viewer.

The Propagator Viewer in Figure 14.2 shows a part of the constraints of a node of the search tree without early failure detection. One may notice the constraints $distinct(\ldots, x_3, \ldots, x_{10}, \ldots) \wedge 1 - x_2 + x_3 = 0 \wedge 1 - x_2 + x_{10} = 0$ (last two lines). Substitution of the two equations yields $x_3 = x_{10}$, which contradicts the constraint $distinct(\ldots, x_3, \ldots, x_{10}, \ldots)$ (top line). Generalization of this observation leads to an

[1]The Propagator Viewer is another application for programming with first-class constraints and originated as byproduct while researching the case studies of this section.

early failure detection criterion: the set δ contains all indices of variables required to be pairwise distinct (derived from the parameters of the *distinct*-constraint). The criterion is: $\exists c_1, c_2 : c_1 \equiv 1 + a_i x_i + a_j x_j = 0 \wedge c_2 \equiv 1 + a_k x_k + a_l x_l = 0 \wedge a_{i,j,k,l} \neq 0 \wedge i = k \wedge j \neq l \wedge j, l \in \delta \wedge a_j = a_l \rightarrow$ failure.

Adding the Early Failure Detection Criterion The early failure detection code is completely factored out. It is embedded in the procedure `DetectFailureEarly` which is applied as soon as constraint propagation reaches a fixed-point, *i.e.*, right before the creation of a new choice-point. Note that Mozart Oz provides means to synchronize on reaching a propagation fixed-point: A procedure can be passed to the search engine (by `FD.distribute` and `FS.distribute`) and this procedure is applied to the solution variable of a search problem as soon as a fixed-point is reached (Section 3.2). The procedure `DetectFailureEarly` reflects the constraints to their first-class representation `Cs` according to a normal form. The variable `EqCs` refers to the equational constraints and the variable `DistinctCs` to the pairwise distinct constraints. Then for each *distinct*-constraint a set δ is computed and stored in the list of sets values `DistinctSets` (see Part II for details on integer sets). Here the implementation is more general than required for this example.

```
proc {DetectFailureEarly RootVars}
   Cs          = {Constraint.reflectSpace RootVars _}
   EqCs        = {FilterEqualityConstraints Cs}
   DistinctCs  = {FilterDistinctConstraints Cs}
   DistinctSets = {ComputeDistinctSets DistinctCs}
```

Then two nested loops (procedures `ForAllTail` and `ForAll` applying anonymous procedures `$`) try to match the appropriate equational constraints according to the early failure detection criterion. An equational constraint is represented by a tuple `'=:'(P LHS RHS)` where `P` is a reference to the actual constraint and `LHS` (`RHS`) is the left hand-side (right hand-side) of the equation. The left resp. right hand-side is represented by a list of addend tuples `addend(Sign Coeff Var)` where `Sign` is the sign (-1 or 1), `Coeff` is the absolute value of the coefficient, and `Var` is a reference to the variable.

Constraints of form $1 + ax + by = 0$ are isolated by pattern matching and the pattern for such a constraint is `'=:'(_ [A1 A2 A3] 0)`[2] as it can be found in the **`case`**-statements.

```
in
   {ForAllTail EqCs
    proc {$ Tail}
       case Tail of  ('=:'(_ [A X1 X2] 0)) | T then
          {ForAll T
           proc {$ TC}
              case TC of '=:'(_ [B Y1 Y2] 0) then
```

After isolating two matching equational constraints the constant addends are compared and it is checked if the variables are in a δ-set. The predicate `Some` is true if at least

[2]Note that there is an order on the addends: the first one is constant, the next ones contain variables and the variables are subject to a certain order.

one of the elements of the list passed (here `DistinctSets`) evaluates the 2nd argument function to **`true`**. The boolean expression e_1 **`andthen`** e_2 is equivalent to **`if`** e_1 **`then`** e_2 **`else false end`**.

```
if A == B andthen
   {Some DistinctSets
    fun {$ Set}
       {VarIsInSet X1 Set}
       andthen {VarIsInSet X2 Set}
       andthen {VarIsInSet Y1 Set}
       andthen {VarIsInSet Y2 Set}
    end}
then
```

Here the anonymous function `$` checks if the variables of the addends are in one and the same δ-set. It uses the predicate `VarIsInSet` which checks if a variable is in a given set. The connector **`andthen`** is a short-circuit conjunction.

```
                  if {IsEqAddend X1 Y1}
                            andthen {IsNeqAddend X2 Y2}
                     orelse {IsEqAddend X1 Y2}
                            andthen {IsNeqAddend X2 Y1}
                     orelse {IsEqAddend X2 Y1}
                            andthen {IsNeqAddend X1 Y2}
                     orelse {IsEqAddend X2 Y2}
                            andthen {IsNeqAddend X1 Y1}
                  then fail % raise failure
                  end
               end
            end % case
         end}
      end % case
   end}
end % DetectFailureEarly
```

Finally, the variables of the addends are tested to meet the early failure detection criterion and if so, failure is raised by the statement **`fail`**. The predicate `IsEqAddend` (`IsNeqAddend`) tests if two addends are equal (not equal). The individual applications of `IsEqAddend` are connected by the short-circuit disjunction **`orelse`**.[3]

Evaluation The evaluation was done on a set of Hamiltonian path problems with the number of nodes ranging from 10 to 50 (it is the same set of problems as used in [61], the problem can be found at [105]). Table 14.1 shows the effectiveness of the presented technique impressively. Entries '-' indicate that after 100.000 nodes of the search tree no solution was found and search was aborted.

By accident the results for problems with 40 and 50 nodes are identical. The first solution was found on a 200MHz Pentium Pro in a range from a tenth of a second till

[3]The boolean expression e_1 **`orelse`** e_2 is equivalent to **`if`** e_1 **`then true else`** e_2 **`end`**.

# nodes	no early failure detection solution found after # choices/# failures	with early failure detection solution found after # choices/# failures	# detected failures
10	72/52	72/52	0
20	-	160/124	1
30	-	298/244	68
40	-	499/406	162
50	-	499/406	162

Table 14.1: Effectiveness of early failure detection.

less than a minute depending on the problem. But the benchmarks aim at demonstrating the effectiveness of the technique, and the early failure detection code has not been particularly optimized.

Early failure detection requires constraints to be first-class citizens in order to reflect the state of the constraint solver for making symbolic detection of inconsistent constraints possible.

14.3.2 Constraint Optimization

This section demonstrates another constraint programming technique made possible by first-class constraints. It is not unusual that a constraint model and consequently its implementation contains redundant constraints or constraints in a formulation that does not allow for the strongest possible propagation.

Consider the constraint $x + x = y \wedge x \in \{1, 2\} \wedge y \in \{3, 4\}$. Without exploiting the aliasing of the two variables on the left hand-side the constraint cannot deduce that the only valid instantiation is $x = 2 \wedge y = 4$. Hence the optimization $x + x = y \rightarrow 2x = y$ improves constraint propagation significantly.

This section reuses the Hamiltonian path problem defined in Section 14.3.1 but uses reified constraints (see page 18) instead of disjunctive combinators. Constraints $c_1, \ldots, c_n$ can be disjunctively connected by reifying them and summing up the $0/1$-variables to 1: $(c_1 \leftrightarrow b_1) \wedge \ldots \wedge (c_n \leftrightarrow b_n) \wedge b_1 + \ldots + b_n = 1$.

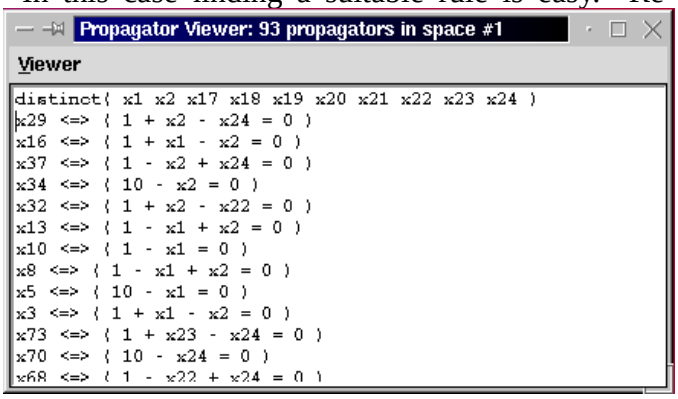
The Constraint Model and its Implementation The constraint model expresses the disjunctions by reified constraints. The parentheses "()" enclosing the equations indicate reification. Constraint (14.5) stands for the path from the starting to the ending node and constraint (14.6) for the same path in reverse direction.

$$distinct(x_1, \ldots, x_n) \tag{14.4}$$

$$\forall\ arc(f, T) \in Arcs : \left((x_f = n) + \sum_{i \in T}(x_i = x_f + 1)\right) = 1 \tag{14.5}$$

$$\forall\ arc(f, T) \in Arcs : \left((x_f = 1) + \sum_{i \in T}(x_i + 1 = x_f)\right) = 1 \tag{14.6}$$

Deriving an Optimization Rule In this case finding a suitable rule is easy. Regard the lines in the figure starting with the variables x_{16} and x_3. In both cases the corresponding constraints reify $1 + x_1 - x_2 = 0$. That makes it possible to equate x_{16} and x_3 and to discard a copy of $1 + x_1 - x_2 = 0$. In general $\exists(c_i \leftrightarrow b_i), (c_j \leftrightarrow b_j) : c_i = c_j \rightarrow b_i = b_j \wedge \mathit{discard}(c_j)$.



The proposed optimization has two effects: it removes redundant propagation by discarding superfluous constraints, and aliases variables.

Adding Constraint Optimization Constraint optimization is executed whenever propagation reaches its fixed-point. It reflects the constraints of a computation space with `Constraint.reflectSpace` to obtain direct access to the constraints, and function `FilterReified` filters out all reified constraints ($c \leftrightarrow b$) since the other constraints are of no interest. The result is stored in `ReCs`. Furthermore, `FilterReified` generates a key for c using `Constraint.getKey` which is used as index for the dictionary `Dict` to easily identify reified constraints which are identical modulo the 0/1-variable b. A dictionary is an abstract data type encapsulating mutable state making it possible to store and retrieve values indexed by keys. Dictionaries are provided by Mozart's base environment [47]. The following operations on dictionaries are used: creating a dictionary (`Dictionary.new`), storing a value indexed by a key (`Dictionary.put`), retrieving a value indexed by a key (`Dictionary.get`), and checking if a key exists (`Dictionary.member`).

```
fun {OptimizeAndCollect RootVars}
   ReCs = {FilterReified
           {Constraint.reflectSpace _ RootVars}}
   Dict = {Dictionary.new}
in
```

For each reified constraint the actual optimization is done in a `ForAll` loop which calls an anonymous procedure `$`. This procedure accesses the components of its argument by pattern matching: `I` is the textual representation index, `P` is a reference to the reified constraint itself, `C` the reified constraint, and `B` is a 0/1-variable. Note that `#` is the infix tuple constructor and hence `I#reified(P C B)` is a 2-tuple matched against the argument passed to the anonymous procedure.

```
   {ForAll ReCs
    proc {$ I#reified(P C B)}
       if {Dictionary.member Dict I} then
          reified(P1 C1 B1) = {Dictionary.get Dict I} in
          B1 = B {Constraint.discard P}
       else
```

```
            {Dictionary.put Dict I reified(P C B)}
         end
      end}
      % return 0/1-variables of the reified constraints
      {RetrieveBools Dict}
   end % OptimizeAndCollect
```

Using `Dictionary.member` the procedure checks if a reified constraint is already stored under the textual representation index `I`. If so, the individual components of the entries are retrieved by pattern matching[4], the 0/1-variables are equated, and the constraint referred to by `P` is discarded by `Constraint.discard`. That is exactly what the optimization rule requires. In case the reified constraint is not yet stored in `Dict` a new entry is created by `Dictionary.put`. Finally, the 0/1-variables of the reified constraints are retrieved and returned by `RetrieveBools`.

The search strategy branches over the 0/1-variables of the reified constraints (14.5) and (14.6) returned by `OptimizeAndCollect` to stay as close as possible to the program used in Section 14.3.1.

Evaluation The evaluation was done on the same set of problems as in Section 14.3.1. The number of 0/1-variables coming from the reified constraints is significantly reduced by optimization. In combination with the additional variable aliasing, this leads to an enormous reduction of choice points (see Table 14.2), even better than for early failure detection in Section 14.3.1.

# nodes	no optimization solution found after # choices/# failures	with optimization solution found after # choices/# failures	with optimization # optimized constraints
10	292/288	4/2	26
20	-	19/0	60
30	-	19/94	118
40	-	2673/2632	158
50	-	122/73	199

Table 14.2: Effectiveness of constraint optimization.

Only for the graph with 40 nodes the number of choice points is much greater. This indicates that the search strategy used is not stable enough against variations of the problems, but this is not the focus of this thesis.

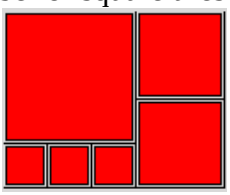
Constraint optimization requires constraints to be first-class values in order to reflect the state of the constraint solver and thus making symbolic constraint optimization possible.

[4]The return value of the function application `{Dictionary.get Dict I}` is matched against the tuple `reified(P1 C1 B1)` and the newly introduced variables `P1` `C1` `B1` are bound accordingly.

14.3.3 Garbage Collection of Constraints

Usually constraint solvers collect redundant constraints as they become entailed by the constraints in the store. Even if their memory is not freed due the implementation of the solver, they are at least not rerun anymore if their parameters receive new basic constraints. For example, consider $x \leq y \wedge x \in \{0, \ldots, 3\} \wedge y \in \{3, \ldots, 6\}$, and suppose the $\leq$-constraint is entailed by the basic constraints $x \in \{0, \ldots, 3\} \wedge y \in \{3, \ldots, 6\}$ and is garbage collected. That is not always the case, as demonstrated for the *nonoverlap*-constraint in the tiling problem.

The Problem Description and the Constraint Model A given number of square tiles has to be placed on a master plate (see figure). The tiles must not exceed the master plate along the x- and y-axis. This is enforced by the *capacity*-constraint which is not of interest here. Furthermore, the tiles must not overlap which is enforced by the *nonoverlap*-constraint. Consider the square tiles T_1 and T_2 with length l_1 and l_2. Their positions on the master plate are determined by their left lower corners (x_1, y_1) and (x_2, y_2) which results in the *nonoverlap*-constraint



$$x_1 + l_1 \leq x_2 \vee x_2 + l_2 \leq x_1 \vee y_1 + l_1 \leq y_2 \vee y_2 + l_2 \leq y_1. \tag{14.7}$$

The constraint (14.7) is encoded by the reified constraint

$$(x_1 + l_1 \leq x_2) + (x_2 + l_2 \leq x_1) + (y_1 + l_1 \leq y_2) + (y_2 + l_2 \leq y_1) \quad \geq 1.$$

Note the $\geq$-constraint which is necessary since two tiles can be non-overlapping in both the x- and y-axis. This constraint causes the trouble regarding garbage collection since as soon as one of its reified constraints is valid the remaining three constraints could be discarded. But this is impossible without first-class constraints because a reified constraint cannot be discarded, it just reduces to its embedded positive or negative constraint.

Implementation Issues The encoding of the *nonoverlap*-constraint by the procedure `ImposeNonOverlap` catches the references to the individual constraints involved, *i.e.*, the reified $\leq$-constraints and the $\geq$-constraint. This is achieved by using the **`prop...end`** statement. The *nonoverlap*-constraint returns these references wrapped in the tuple `nonoverlap(P0 [P1 P2 P3 P4])`.

```
fun {ImposeNonOverlap X1 Y1 L1 X2 Y2 L2}
   B1 B2 B3 B4 P0 P1 P2 P3 P4
in
   prop {[P1]} (X1 + L1 =<: X2) = B1 end
   prop {[P2]} (X2 + L2 =<: X1) = B2 end
   prop {[P3]} (Y1 + L1 =<: Y2) = B3 end
   prop {[P4]} (Y2 + L2 =<: Y1) = B4 end
   prop {[P0]} B1 + B2 + B3 + B4 >=: 1 end

   nonoverlap(P0 [P1 P2 P3 P4]) % return value
end
```

The procedure `CollectNonOverlapConstraints` is called when propagation has reached a fixed-point. It receives as its argument a list of tuples produced by `ImposeNonOverlap` and checks for all tuples if the enclosed $\geq$-constraint is entailed by applying `Constraint.isEntailed` to `P0`. If so, the remaining reified constraints are discarded by `Constraint.discard`.

```
proc {CollectNonOverlapConstraints NonOverlapConstraints}
   {ForAll NonOverlapConstraints
    proc {$ nonoverlap(P0 Ps)}
       if {Constraint.isEntailed P0} then
          {ForAll Ps proc {$ P}
                        if {Constraint.isEntailed P} then skip
                        else {Constraint.discard P} end
                     end}
       end
    end}
end
```

Evaluation Table 14.3 shows the number of reified $\leq$-constraints garbage collected for different instances of the tiling problem.

# tiles	# collected constraints	saved memory
6	18	5K
9	44	11K
17	197	100K
21	1528	1.7M

Table 14.3: Effectiveness of constraint garbage collection.

The third column shows the amount of memory saved which is in balance with the overhead imposed by the extra data structures used.

The proposed garbage collection scheme relies on first-class constraints for detecting redundant constraints and for explicitly discarding such constraints since they cannot be garbage collected yet by entailment.

14.3.4 Smallest Sets of Inconsistent Constraints

First-class constraints can be used to find smallest subsets of inconsistent constraints of a given set of inconsistent constraints (for short SIC). In this section, a set of constraints is called *inconsistent* if constraint propagation causes a failure.[5] A SIC I is the *smallest* if as soon as any $c_i \in I$ is removed from I, no failure is caused anymore by constraint propagation. Note that there may be more than one smallest SICs which are pairwise mutually not inclusive.

[5]It does not mean that a set of constraints which is *not* detected as inconsistent by constraint propagation is consistent.

Smallest SICs are interesting for debugging constraint programs. For example, Ueda *et al.* use smallest SICs in [30, 5, 4] to spot the reason for programs not being well-moded *resp.* well-typed. Another example is that the first run of a constraint program frequently results in an immediate failure of the solver. This is caused by a SIC which is usually large. But only a subset causes the failure and hence, being able to find smallest inconsistent subsets of a given SIC may simplify debugging significantly.

Algorithm The algorithm for finding smallest SICs consists of two parts: part (i) finds an initial smallest SIC and part (ii) finds another smallest SIC which is mutually not inclusive with all already found smallest SICs. Part (ii) of the algorithm is iterated until no further smallest SICs are found.

Ueda *et al.* give an algorithm[6] for finding a smallest sub-SIC for a given SIC $I = \{c_1, \ldots, c_n\}$ [30, 5, 4]. This algorithm requires k iterations over the n constraints where k is the number of constraints in the smallest SIC. This results in an asymptotic complexity of $\mathcal{O}(k \times n \times f(n))$ if checking consistency is assumed to have a complexity of $f(n)$. The algorithm in Program 14.1 needs exactly one iteration over the c_i, *i.e.*, it runs in $\mathcal{O}(n \times f(n))$. This algorithm speculatively removes every constraint c_i and checks if the resulting constraint in I' is still inconsistent. The function application *is_inconsistent*(I') performs consistency checking on I' (for example by running the consistency algorithm in Program 2.1 on page 10)

```
I = {c_1, ..., c_n}
for (i = 1; i ≤ n; i = i + 1) {
  I' = I \ {c_i}
  if (is_inconsistent(I'))
    I = I'
}
```

Program 14.1: Finding a smallest set of inconsistent constraints.

Part (ii) of the algorithm searches for new smallest SICs by activating *resp.* deactivating individual constraints c_i. Branch-and-bound search uses the ordering constraint (14.8) to find a smallest SIC (O is the current SIC and N denotes a new SIC).

$$N \subset O \tag{14.8}$$

Let $\mathcal{P}$ denote the set of previously found smallest SICs P_i. To ensure that the new subset N is mutually not inclusive to the already found smallest SICs $P_i \in \mathcal{P}$, the constraint (14.9) is added to the search problem:

$$\forall P_i \in \mathcal{P} : P_i \setminus (P_i \cap N) \neq \emptyset \ \wedge \ N \setminus (P_i \cap N) \neq \emptyset \tag{14.9}$$

Note that for every iteration of part (ii) of the algorithm, $\mathcal{P}$ is updated with the new smallest SIC N.

[6] A dedicated algorithm for finding efficiently an initial subset is needed since using naïve search including *resp.* excluding constraints c_i has exponential complexity.

Implementation Constraints in a SIC are implemented by first-class propagators. The idea of the implementation is to connect the first-class propagators with 0/1-control variables. Constraining a control variable to 0 discards the connected propagator while constraining the variable to 1 activates the propagation of the connected propagator. Procedure `ImposeConstraint` connects a propagator with a control variable.

```
proc {ImposeConstraint P B}
   prop inactive {[C]} {P} end
   B :: 0#1
   thread
      if B == 1 then {Constraint.activate C}
      else {Constraint.discard C} end
   end
end
```

The first parameter `P` of `ImposeConstraint` is a procedure[7] which imposes a propagator with deactivated propagation. Applying `P` within a **`prop ... end`** statement binds `C` to the corresponding first-class propagator. The **`if`**-statement is run on a separate thread to avoid the execution of the procedure `ImposeConstraint` blocking.

A SIC is represented by a finite set of integers. Every propagator p representing a constraint in the SIC I is identified by a unique integer i_p. The subset of inconsistent constraints is the set of identifiers i_p represented by a finite set of integers (see Part II). Adding i_p to I turns propagation of p on and constrains the control variable of p to 1. Removing i_p from I turns propagation of p off and constrains the control variable of p to 0. This works also in the other direction (similar to reified constraints) by binding the control variable.

Part (i) of the algorithm implements Program 14.1. The application of the function *is_inconsistent*() is encapsulated by the negation combinator (Section 3.3.2) to catch and detect failure. The result is a set of integer identifiers referring to the constraints contained in the initial smallest set of inconsistent constraints.

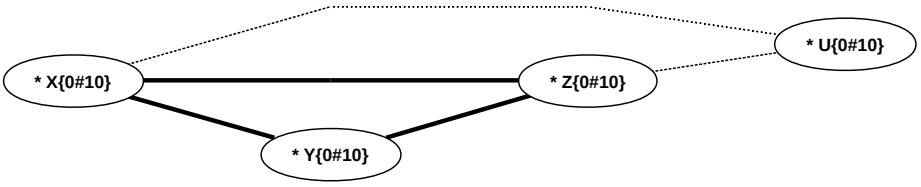
Part (ii) of the algorithm implements branch-and-bound search using Mozart's search facilities (Section 3.3.1). The search problem consists of a SIC I, the constraint (14.9), and the ordering constraint (14.8). Again, the SIC is encapsulated by the negation combinator.

Distribution over first-class propagators is done by distributing over their 0/1-control variables using the finite domain library abstraction `FD.distribute`. The constraints (14.8) and (14.9) can be straightforwardly expressed by finite set constraints (Part II).

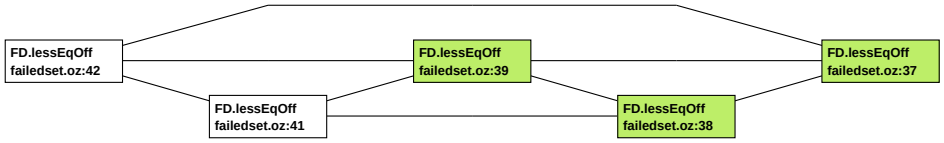
Example Consider the inconsistent set of constraints $\{x < y, y < z, z < x, z < u, u < x\}$. As one can easily see, there are two different smallest sets of inconsistent constraints: $I_1 = \{x < y, y < z, z < x\}$ and $I_2 = \{x < y, y < z, z < u, u < x\}$. As expected, the search routine finds two smallest SICs I_1 and I_2.

The first solution, corresponding to I_1, is shown as a graph in Figure 14.3(a) where

[7] An example for `P` is **`proc`**`{P} {FD.lessEqOff A B C}` **`end`** where the propagator application `{FD.lessEqOff A B C}` imposes the constraint $A \leq B + C$.



(a) Parameter graph where failed propagator edges are thick solid line.



(b) Constraint graph where nodes of failed propagators are shaded.

Figure 14.3: First solution I_1.

the nodes of the graph denote variables and edges represent propagators. Thick solid edges stand for propagators that are part of the SIC. Figure 14.3(b) depicts SIC I_1 as graph where nodes denote propagators and edges denote variables shared between propagators. Propagators that are part of the SIC are shaded. The second solution, corresponding to I_2, is shown in Figure 14.4.

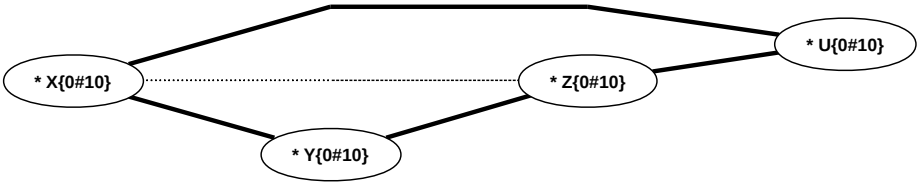
The graphs in Figures 14.3 and 14.4 were generated by the interactive constraint debugging tool Investigator which is presented in Chapter 15. The annotations `FD.lessEqOff` of the nodes in Figure 14.3(b) and Figure 14.4(b) refer to the propagators which realizes the $<$-constraint. The Investigator can be used to debug an immediately failing constraint program: after the propagators responsible for failure are identified, their occurrence in the source code can be easily spotted by clicking on the corresponding propagator nodes in the Investigator (see Figure 15.6 on page 173).

First-class constraints are the key to this application since with them one can search over constraints by explicitly turning propagation on and off.

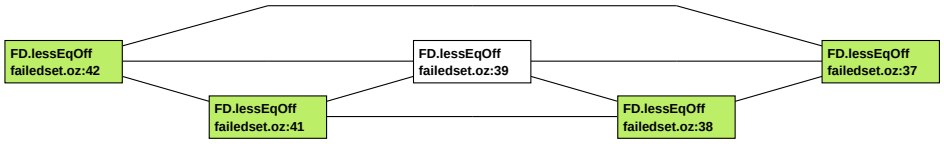
14.4 Implementation

Adding first-class propagators to the propagation services presented in Part I requires only conservative extensions which do not impose any performance penalties if first-class propagators are not used.

Propagators as Values A first-class propagator is a reference to a propagator and is represented by a new class of heap objects. The operations are passed on via the first-class



(a) Parameter graph where failed propagator edges are thick solid line.



(b) Constraint graph where nodes of failed propagators are shaded.

Figure 14.4: Second solution I_2.

propagator to the actual propagator which implements the operation by corresponding functions. A first-class propagator is created by an extended creator function which imposes the actual propagator and returns the first-class propagator by an extra argument. The **`prop...end`**-operator directs the compiler to call such extended creator functions instead of the standard ones.

Controlling Constraint Propagation Constraint propagation of a propagator is controlled by an extra flag which indicates whether propagation is turned on or turned off. A deactivated propagator is cannot be scheduled, *i.e.*, it remains in the execution state *`sleeping`*. If a propagator's propagation is turned on, it is immediately scheduled.

Entailment A propagator is entailed by setting its execution state to *`entailed`*. Nothing else needs to be done. To find out whether a propagator is entailed or not simply requires to check the execution state of the propagator.

Reflection Obtaining the name and the parameters of a propagator is already supported by the corresponding functions of a propagator. The implementation of the reflection of a whole computation space (`Constraint.reflectSpace`) maintains a set of propagators S_P and a set of variables S_V. First, it adds the given variables in `Rs` to S_V and adds the propagators in the event lists of the variables in S_V to S_P. It proceeds by adding the parameters of newly found propagators to S_V and the propagators in the event lists of newly found variables to S_P. Thus the whole constraint graph is traversed until no new propagators and variables are found. Detecting the membership of a variable in S_V or a propagator in S_P has to be efficient. Hence, the sets S_V and S_P are implemented by hash tables.

Identification First-class propagators can be identified to refer to the same propagator simply comparing their references. The identification of individual variables in a list of variables collects the variables in a dictionary data structure. The keys of the dictionary are the locations of the variables. A variable newly added to the dictionary is assigned an incremented counter value while an already contained variable takes the assigned integer as identifier.

14.5 Related Work and Discussion

Some of the programming techniques discussed in this chapter have been successfully applied in the area of constraint-supported proof planning as presented in [93, 94]. The benefit of first-class constraints was the ability to concurrently perform domain and symbolic reasoning.

The idea to combine symbolic and domain reasoning has been used by Hong to implement solver over complex functions [69] and non-linear constraints over real numbers [68]. The solvers have been implemented using computer algebra systems. First-class constraints seem to be suitable for this kind of constraint solvers but this has to be proved by future research (see Section 16.2).

One approach at gaining more control and expressiveness over constraints was the idea to exploit a constraint's truth value as proposed for the cardinality constraint in [143]. Applying arithmetic and boolean operations to constraint's truth values was explored in [12]. These constraints are usually called meta or reified constraints. They are available in nearly all current constraints solvers.

Meta-programming as known from Lisp or Prolog means manipulating a program by another program. Therefore, the program code is represented as a term of the respective programming language and then submitted to a meta-interpreter written in this language. Such a scheme for the constraint programming language CLP($\mathcal{R}$) is proposed in [62]. They use `quote` and `eval` functions which are analogous to the corresponding Lisp functions.

Solvers dedicated to a certain set of constraints can of course do the same analysis as discussed in this chapter. Harvey and Stuckey describe in [61] a propagation based solver for linear (in-, dis-)equations which is able to detect failure as early as the approach described in Section 14.3.1.

ILOG SOLVER [73] is a C++ library for constraint programming in C++. It does not support first-class constraints as presented but ILOG SOLVER 5.0 allows the user to define a new constraint by defining a new class of constraints derived from the library class `IlcConstraintI`. It is straightforward to provide the required extra functionality according to Section 14.4 by adding appropriate member functions to the class definition of the new constraint.

Constraint Handling Rules (CHR) [51] are a committed-choice language for rewriting constraints towards a solved form which eventually denotes a solution. A CHR program is a set of guarded rules of the form $H \quad op \quad G \mid B$ where $op \in \{$**<=>**, **==>**$\}$, $H = H_1, \ldots, H_i$, $G = G_1, \ldots, G_j$, and $B = B_1, \ldots, B_k$. A multi-head H is a se-

quence of CHR, the guard G is a sequence of built-in constraints, and the body B is a sequence of CHR and built-in constraints. A rule fires as soon as a the CHR store implies H and the constraint store implies G. Then the CHR and constraint store are extended by B. A propagation rule ($op =$ `==>`) extends the appropriate stores by redundant constraints B. A simplification rule ($op =$ `<=>`) behaves like a propagation rule but additionally removes H from the CHR store. CHR can be used to implement the techniques proposed in Section 14.3.1 and Section 14.3.2 due to the multi-heads of the rules. For example, the inconsistent constraint $x < y \wedge y < x$ can be detected by the following CHR rule: `less(x,y),less(y,x) <=> true | false`.

The programming techniques presented in Section 14.3 need to detect the fixed-point of constraint propagation. Hence the constraint solver has to provide means that allows the programmer to synchronize computation with reaching a propagation fixed-point. The implementation of such a check is straightforward and simply tests the number of runnable propagators.

None of the above-mentioned approaches offers the same expressiveness or generality as the scheme proposed in this thesis, to promote constraints to first-class citizens.

Chapter 15

Debugging Constraints

This chapter develops a debugging scheme based on a graph view metaphor. Based on this scheme, an interactive debugging tool, called Constraint Investigator, is presented and demonstrated by an example debug session with a prototype implementation of the Investigator[1]. This implementation was intended to successfully prove the concept of the debugging scheme and is another case study that demonstrates the benefits of using first-class constraints in programming.

15.1 Overview

Propagator-based constraint solvers are able to tackle large instances of combinatorial problems. But developing solvers for such problems has only limited support by debugging tools. This deficiency has been identified and dedicated projects (as DiSCiPl [41, 37]) have been set up.

The first step to be taken when solving a combinatorial problem is to design a constraint model of the respective problem, *i.e.*, to find a problem formulation in terms of constraints. Next this model is implemented by some constraint solver. Testing the implementation reveals quite frequently that no solution can be found, the solution found is not correct, or the solution found still contains undetermined variables. These situations suggest that the constraint model or its implementation do not reflect the combinatorial problem to be solved. To support the development process at this stage, the programmer needs adequate interactive debugging tools which are currently not available.

Current constraint debugging tools focus on improving search behavior (*e.g.*, [126, 91, 134]), *i.e.*, on finding most suitable branching and exploration algorithms. There is a lack of intuitive interactive tools for debugging the correctness of constraint models and/or their implementations. In particular, large problems need tools with a sophisticated presentation to handle the overwhelming amount of information. Hence, providing an appropriate metaphor to present the data is crucial. The model of data presentation proposed in this thesis is derived from graph-based visualization, as proposed by Carro

[1] The source code of the Investigator can be obtained via [105].

and Hermengildo in [25]. The graph metaphor was first formally introduced in constraint programming by Montanari and Rossi in [96].

This chapter develops different graph-based views for correctness debugging constraint programs and a debugging methodology based on these views for frequently occurring incorrect behavior of constraint programs. Further, techniques for handling large problems are proposed.

The viability of the approach is proved by designing and prototypically implementing an interactive tool, the Constraint Investigator, that allows the user to investigate the state of constraints and variables in a constraint solver by analyzing the corresponding graph views. The Investigator is characterized by the following points:

- It is not restricted to any specific constraint domain.
- It relies on a propagation-based constraint solver.
- It provides intuitive data presentation and interaction, while affording detailed insights about the solver.
- It is fully configurable by the user and requires no changes to the actual constraint program.
- It is suitable for users at different levels of expertise.
- It reveals operational aspects of the solver by displaying the events that trigger constraints.

A prototype of the Constraint Investigator is implemented in Mozart and the visualization of the graph views relies on *daVinci* [50, 36]. The Investigator complements the Oz Explorer as plug-in. Both tools form the base of an integrated constraint debugging environment.

The Constraint Investigator can be also useful for performance debugging. For example, its graph views can be aumented with execution costs of constraints such that the program code causing these costs can be identified. Furthermore, operational aspects of constraint execution (see Section 2.3 about events) are revealed and can be used to improve execution performance.

15.2 Debugging Constraints

Debugging an application focuses first on correctness and then on performance. Approaches to debugging can be identified as *experimental* and *analytic* [91]. Experimental debugging, *i.e.*, modifying the program text until it seems to work, requires a large set of methods to experiment with. In contrast, analytic debugging needs to obtain a detailed description of the state of the constraint solver. Such a description has to be presented to the programmer by a debugging tool in a way that supports program analysis in the best possible fashion.

After designing and implementing the constraint model of a given problem, testing the implementation typically produces erroneous situations as:

- The solver fails immediately, *i.e.*, the constraints are inconsistent. Either the implementation of the constraint model is incorrect or the model itself is. It is often

the case that by accident the constraint model is over-constrained though the combinatorial problem is not. For example, the model states an equivalence where an implication is required. In such a case, if a solution is available (perhaps manually derived), it is a promising strategy to debug this situation by adding this solution to the constraint statements. The propagator which is observed to fail is not necessarily the culprit for the bug in the implementation but it helps to track down the problem in the constraint model.
- Propagation is incomplete in the sense that some solution variables remain undetermined. This is an indicator that the implementation or the model is incomplete.
- The solution found is wrong. Either the constraint model is incorrect or if this is not the case, the implementation of the model is incorrect.

The proposed debugging approach and the corresponding tool are aimed at analytic correctness debugging, *i.e.*, to spot bugs in the constraint model and its implementation.

Analytic debugging requires an interactive tool that enables the programmer to analyze the actual constraints in the solver. The amount of information, *i.e.*, typically the number of variables and constraints, is huge. The way these data are presented in analytic debugging is important since constraint programs are data-driven and an appropriate presentation helps the programmer to draw the right conclusions. Hence, data representation has to match the programmer's intuition of constraints in a constraint solver. Consequently, the graph-based metaphor is chosen for representation since it makes it possible to emphasize different aspects of the state of a constraint solver appropriately (see the different views presented in Section 15.3) and to relate the program structure to the representation (see Section 15.4.2).

15.3 Graph-based Visualization of Constraints

In this section, different graph views are illustrated using a trivial scheduling application. The problem is to serialize two tasks, such that they do not overlap. The first (second) task starts at starting time *T1* (*T2*) and has a fixed duration of *D1* (*D2*). The corresponding constraint model is the disjunction $T1 + D1 \leq T2 \vee T2 + D2 \leq T1$. The concrete implementation uses reified constraints to implement the disjunction. A reified constraint has an extra boolean parameter that reflects the validity of the constraint, *i.e.*, whether it is entailed or failed. For example, $B1 = (T1 + D1 \leq T2)$ is the reified version of $T1 + D1 \leq T2$ and if this constraint is entailed (failed) *B1* is bound to 1 (0). Conversely, in case *B1* is bound to 1 (0) the constraint $T1 + D1 \leq T2$ ($T1 + D1 > T2$) is stated. The (exclusive) disjunction of the constraints can be implemented by stating that the sum of the boolean variables associated with the reified constraints is 1. The following Oz code implements the serialization constraint for two tasks[2]:

```
B1 =: (T1 + D1 =<: T2)    % implemented by FD.reified.sumC
B2 =: (T2 + D2 =<: T1)    % implemented by FD.reified.sumC
```

[2]Note that `D1` and `D2` refer to integers and all other variables are finite domains. The $=$-constraint is implemented by Oz's finite domain operator `=:` (and $\leq$ by `=<:`).

```
B1 + B2 =: 1                    % implemented by FD.sumC
```

Four different views of the above constraint program are presented. The shape of a node represents its kind: a propagator node is a rectangle, a variable node is an ellipse, and an event node is a rhombus. A propagator node is annotated with the name of the respective propagator and the location of the propagator invocation in the source program, *i.e.*, the file name and the line number. A variable node is annotated with the name of the respective variable and if the variable is constrained, the basic constraint connected to the variable is also shown. Note that there are no variable nodes for `D1` and `D2` since they denote integers.

Preliminaries The constraints of a problem instance can be regarded as a network of propagators P, variables V, and events E. The variables in V are the parameters of the propagators in P. The events in E denote the changes to the basic constraints that trigger a propagator to be scheduled. A propagator $p(v_1^{e_1 \in E_p}, \ldots, v_n^{e_n \in E_p})$ has a set of parameters $V_p = \{v_1, \ldots, v_n\} \subseteq V$ and is triggered by the events $E_p \subseteq E$. The notation $v_i^{e_i \in E_p}$ means that the propagator p is scheduled as soon as event e_i occurs at parameter v_i. A variable $v(p_1^{e_1 \in E_v}, \ldots, p_m^{e_m \in E_v})$ is a parameter of the propagators $P_v = \{p_1, \ldots, p_m\} \subseteq P$ and changes to the basic constraint at v can cause the events $E_v \subseteq E$. The notation $p_i^{e_i \in E_v}$ means that the propagator p_i is scheduled as soon as event e_i occurs at the variable v.

The Propagator Graph View A propagator graph (Figure 15.1) is the graphical representation of a propagator net, *i.e.*, the propagators are the nodes. Note that the edges are not directed since data flow between propagators is bidirectional. This, for example, is different for a constraint solver using indexicals [32] because an indexical is a function rather than a relation. For instance, the leftmost node corresponds to the propagator `FD.sumC` which happens to occur at line 260 of file `opi.oz` (the location of `FD.sumC` while producing the example graph views). This annotation depends on the concrete location of a propagator in a source file. An edge between two nodes means that the propagators share at least one variable parameter.

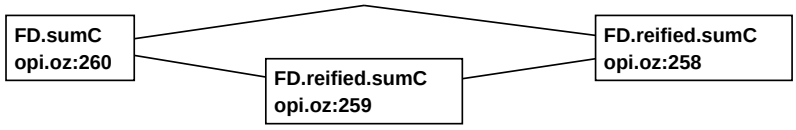


Figure 15.1: A Propagator Graph View.

Using the sets P, V, and E defined in this section, a propagator graph $pg(P_{pg})$ consists of nodes $N_{pg} = P_{pg}$ and edges $E_{pg} = \{(p_i, p_j) | V_{p_i} \cap V_{p_j} \neq \emptyset \wedge i < j\}$.

The Single Propagator Graph View A single propagator view (Figure 15.2) presents a single propagator and its parameters as a tree. The parameters are grouped by the events.

Note a variable may occur several times as parameter. The single propagator graph view of `FD.reified.sumC` shows that the propagator waits for two events, namely the bounds-event, *i.e.*, the bounds of the domain are narrowed, and the any-event, *i.e.*, an arbitrary element is removed from the domain. Furthermore, the view shows that a bounds event at the parameters `T1` resp. `T2` and an any event at `B1` cause the propagator to be scheduled. A variable node is annotated, as for example the node for `T1`: `*T1{0#5}`. This means that `T1` takes a value from $\{0, 1, 2, 3, 4, 5\}$. The asterisk (`'*'`) denotes a variable passed directly by the user to the Investigator in contrast to variables collected while traversing the constraint network.

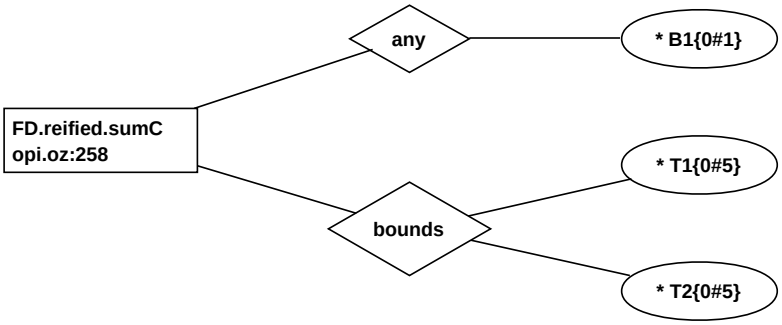


Figure 15.2: A Single Propagator Graph View.

More formally, a single propagator graph $spg(p)$ for a propagator p is a tree with a root node $R_{spg} = p$, connected to the root node are event nodes $E_{spg} = E_p$ and connected to the event nodes variable nodes $V_{spg} = V_p$. An edge between an event node and a variable node is established if the events of the event node and variable node are the same.

The Variable Graph View A variable graph view (Figure 15.3) is dual to the propagator graph view. The nodes represent the variables. An edge between two variable nodes indicates that the variables are simultaneously constrained by one or more propagators. The information of what propagators are concerned is available by a menu associated with the edge. The variable graph view shows that in the example, all variables are connected with each other.

The formal description of a variable graph makes the duality to a propagator graph obvious: a variable graph $vg(V_{vg})$ is composed by the nodes $N_{vg} = V_{vg}$ and the edges $E_{vg} = \{(v_i, v_j) | P_{V_i} \cap P_{V_j} \neq \emptyset \wedge i < j\}$. An edge between two variable nodes is present if the respective variables share at least one propagator.

The Single Variable Graph View A single variable graph view (Figure 15.4) represents a constraint variable, events it can cause and the propagators waiting for these events to happen. One can see that the two reified propagators wait for the bounds event and no propagator waits either for the any event nor for the val event.

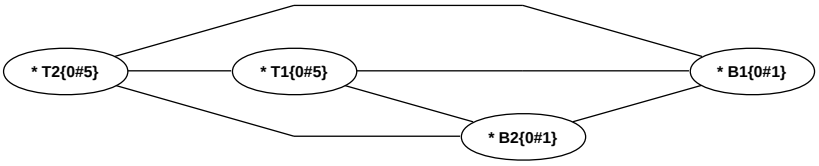


Figure 15.3: A Variable Graph View.

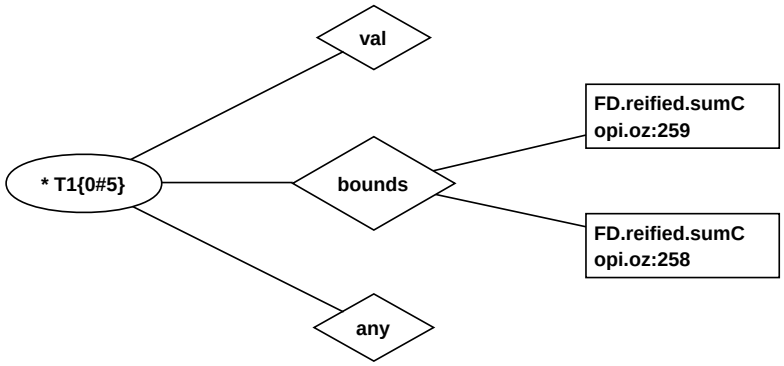


Figure 15.4: A Variable Graph View.

A single variable graph $svg(v)$ of a variable v is a tree with a root node $R_{svg} = v$. Event nodes $E_{svg} = E_v$ are connected to the root node. Furthermore, each event node of an event e is connected to the propagator nodes $P^e_{svg} = \{p^e | p^e \in P_v\}$, *i.e.*, an edge between an event node and a propagator node is established if the propagator waits for this event to happen to this variable.

15.4 Correctness Debugging with the Constraint Investigator

This section introduces the *Constraint Investigator* as an interactive tool for debugging practical constraint problems. Using the Investigator does not require any changes to the constraint program. The program has to be recompiled with appropriate compiler switches.

15.4.1 An Example Session with the Investigator

This section starts off with a deliberately buggy constraint model and program and demonstrate how to track down two hidden bugs. Of course, the bugs are trivial to fix

for experienced programmers but the approaches demonstrated are suitable for handling real-life situations.

The Problem Consider the following bin-packing problem: a given set of weighted items I has to be assigned to three bins $b_{1,2,3}$, without exceeding the maximum capacity of each bin. All bins have the same maximum capacity c. Furthermore, as soon as at least two items are put into a bin one extra unit of packaging material must be added as protection. Moreover, the bins must be color-coded to indicate the presence of a fragile item.

The Constraint Model The given problem is a set partitioning problem of three sets with extra constraints. Each bin b_n is modeled as set s_n and each item $i \in I$ has a weight w_i.

$$I = \uplus s_n \quad (1) \qquad |s_n| \geq 2 \leftrightarrow \text{packaging material} \in s_n \quad (2)$$
$$\Sigma_{\forall i \in s_n} w_i \leq c \quad (3) \qquad i_{fragile} \in s_n \rightarrow color(s_n) = red \quad (4)$$
$$\text{where } n = 1, 2, 3.$$

(1) states a set partitioning and (2) adds extra packaging if necessary. Furthermore, (3) enforces that the capacity of the bins is not exceeded and takes also into account packaging material added by (2). The coloring of the bins is modeled by (4). The model is not quite correct as will become clear later on.

The Implementation of the Constraint Model The implementation of the presented model is based on finite set constraints (Part II), *i.e.*, a set value is approximated by a lower bound set and a upper bound set. The constraint solver has been implemented by the procedure `BinPacking`:

```
proc {BinPacking Weights Capacity Sol}
```

The argument `Weights` is a list of pairs `Id#Weight`. The variable `Capacity` determines the maximum capacity of the bins. The solution is returned in `Sol` and contains the colored bins with the assigned items.

The procedure starts with variable definitions: it declares the variables `Red` and `Green` for the bin-coloring constraint for the fragile item defined by `Fragile`. Next, it adds for the packaging material an extra item (`Packaging=100`) with weight 1 to the list of all weighted items `AllWeights`. The list of `Items` is extracted from the weight list (`AllWeights`).

```
   Red = 0  Green = 1  Fragile = 1  Packaging = 100
   WeightedPackaging = [Packaging#1]
   AllWeights = {Append WeightedPackaging Weights}
   Items = {Map AllWeights fun {$ E} E.1 end}
in
```

The body of the procedure starts by creating the solution list `Sol` of length 3. Each list element represents a bin as a record `bin(items:S color:C)` where `S` is the set of items and `C` is the color of the bin. The application of `{FS.var.upperBound Items}` constrains `S` to the set constraint $\emptyset \subseteq$ `S` $\subseteq$ *setof*(`Items`).

```
   Sol = {List.make 3}
   {ForAll Sol fun {$} S = {FS.var.upperBound Items} C in
```

```
                C :: [Red Green]
                bin(items: S color: C)
             end}
```

Next the partitioning constraint is stated (`FS.partition`). The `Map` function extracts the sets that form the partition from the bin records. The variable `Items` is converted to a set value by `FS.value.make` representing the set to be partitioned.

```
% constraint (1): partitioning
{FS.partition
 {Map Sol fun {$ S} S.items end} {FS.value.make Items}}
```

The weight restriction constraint maps the presence of elements to the list of boolean variables `BL` by `FS.reified.areIn`. The constraint `{FD.sumC ... '=<:' ...}` enforces that the scalar product of the list of boolean variables `BL` and the corresponding list of weights (produced by `Map`) does not exceed `Capacity`.

```
% constraint (3): enforce weight restriction in bins
{ForAll Sol proc {$ S} BL in
             {FS.reified.areIn Items S.items BL}
             {FD.sumC {Map AllWeights fun {$ E} E.2 end}
             BL '=<:' Capacity}
          end}
```

The constraints for adding packaging material and assigning the bin color close the procedure and use reified constraints. Reified propagators are used to conditionally state constraints according to (2) in the constraint model. As soon as the cardinality of `S.items` is at least 2 the item `Packaging` is added to `S.items`. This is caused by the connection through the boolean variables of the reified constraints.

```
% constraint (2): add extra packaging material
{ForAll Sol proc {$ S}
             ({FS.card S.items} >=: 2) =:
              {FS.reified.include Packaging S.items}
          end}
```

The constraint for coloring the bins also uses reified constraints and implements the "$\rightarrow$"–operator of (4) by the implication constraint `FD.impl`[3].

```
% constraint (4): assign colors to bins
{ForAll Sol proc {$ B} {FD.impl
             {FS.reified.include Fragile B.items}
             (Red =: B.color) 1}
          end}
end % BinPacking
```

The code for controlling search is omitted since it is not of interest here and an adequate search strategy is assumed. Now the bin-packing solver is submitted to a search engine, like the Oz Explorer (see Figure 3.6 in Section 3.3.1):

```
{ExploreOne {BinPacking [1#3 2#2 3#2 4#6 5#2 6#4 7#3 8#5] 10}}
```

[3]This is a reified constraint such that the last parameter `1` is required.

This results in an immediately failed search tree. The Investigator is now demonstrated in a prototypical debugging session.

The Implementation is not Faithful to the Constraint Model Invoking the Investigator from the failed node switches the Investigator to the single propagator graph view (see Figure 15.5). The node representing the failed propagator is colored red throughout the session.

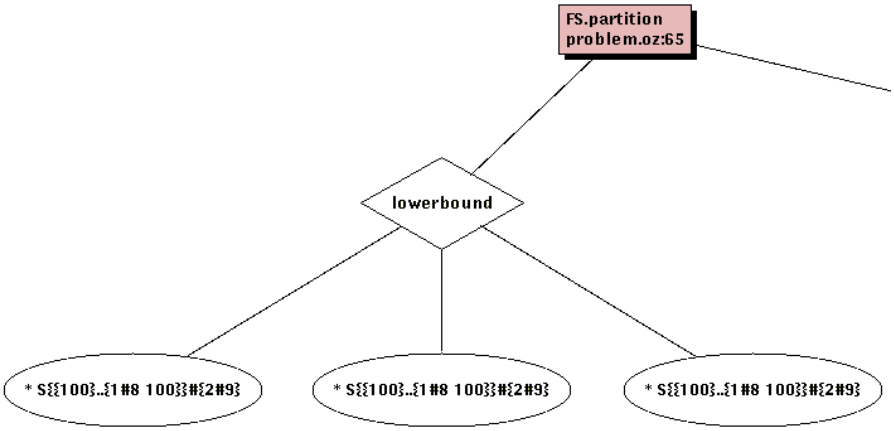


Figure 15.5: Single propagator view of the failed propagator `FS.partition`.

The single propagator graph view in Figure 15.5 shows the partition propagator with its parameters connected via the lowerbound event. The parameters are set constraint variables and are represented by `S{{100}..{1#8 100}}#{2#9}`[4]. This corresponds to the basic constraint $\{100\} \subseteq S \subseteq \{1, \ldots, 8, 100\} \wedge 2 \leq |S| \leq 9$. One can see that all three parameters contain at least element 100. Hence, the partitioning propagator must fail. This reveals an incorrectness but this is not necessarily the actual bug. A single click on the propagator node highlights the line of source code where the partitioning propagator is stated (Figure 15.6).

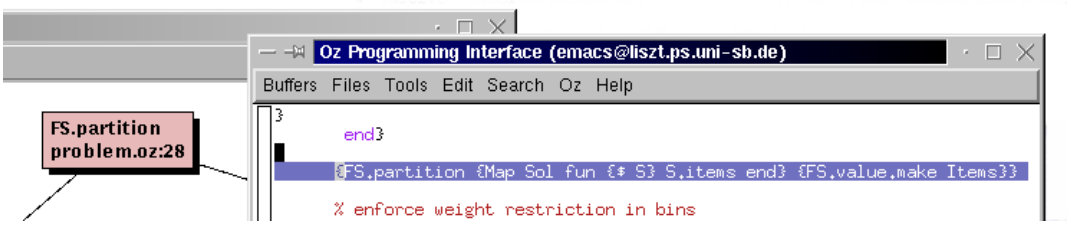


Figure 15.6: Associating the failed propagator to the source program.

[4]That all variables have the same name `S` does not mean that they are aliased. The name is derived from the source code of constraint (1), *i.e.*, `{FS.partition {Map Sol` **`fun`** `{$ S} S.items` **`end`**`} ...}`.

The parameters concerned are the sets of items for each of the bins in the solution `Sol`. Checking the program text suggests that only the implementation of the packaging (3) adds to all item fields of `Sol` the element `Packaging` (which is 100). Verifying the code for adding extra packaging material reveals the bug in the implementation: instead of using different packaging material for each bin, the same material is used for all bins. This is not the intention of the constraint model and hence an implementation bug. The bug fix simply consists of using different packaging material items for each bin and modifies the `ForAll` – loop[5] to select for different bins different packaging material.

```
% packaging material for every bin
WeightedPackaging =
  [(Packaging+1)#1 (Packaging+2)#1 (Packaging+3)#1]
...
{List.forAllInd Sol
 proc {$ I S} % 'I' counts from 1 to length of 'Sol'
    % select different packaging material by the index I
    ({FS.card S.items} >=: 2) =:
        {FS.reified.include 100+I S.items} end}
```

After fixing the implementation bug,

```
Sol = [bin(color:0      items:{1#3 5 101}#5)
       bin(color:_{0#1} items:{4 7 102}#3)
       bin(color:_{0#1} items:{6 8 103}#3)]
```

is obtained as solution and where not all variables are bound to a single value (observe the `color` fields). The next section demonstrates how to track down the reason for this problem.

Identification of Remaining Propagators A solution with unbound variables suggests that there is a lack of propagation. The variable graph view shown in Figure 15.7 is produced when starting the Investigator from the solution node of the Explorer.

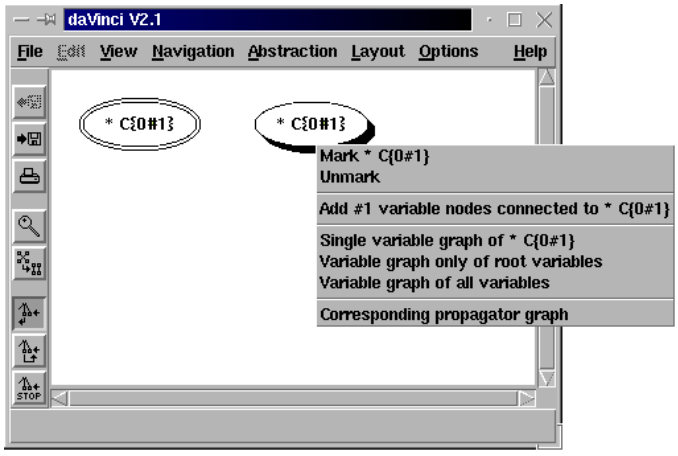
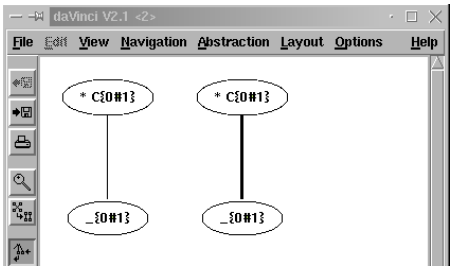


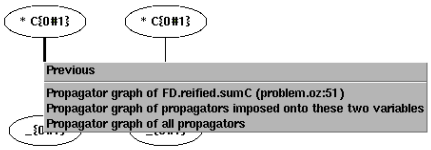
Figure 15.7: Initial view.

[5] `{List.forAllInd [X1...Xn] P}` ≡ `{P 1 X1} {P 2 X2}... {P n Xn}`

The variable `Sol` is not displayed because it is bound to the solution list and hence no variable anymore. One can find remaining propagators starting from one of the variable nodes. Assume one decides to switch to the variable graph view of all reachable variables (Figure 15.8(a)), to get an overview over all variables left unbound. The menu associated with an edge between two variable nodes (Figure 15.8(b)) offers to switch to a single propagator graph view of a propagator being imposed upon two variables.



(a) Variable graph view of all reachable variables.



(b) Edge menu of the variable graph view.

Figure 15.8: Variable graph view.

Since remaining propagators are to be found, it makes sense to switch to the offered single variable graph view of a reified sum propagator (Figure 15.9).

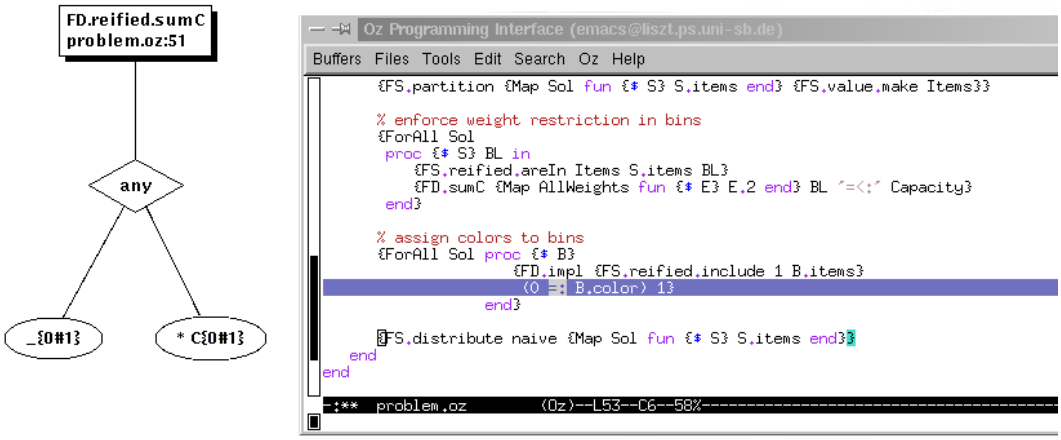


Figure 15.9: Single propagator graph view.

A click on the propagator node immediately reveals the suspicious program text: the assignment of the bin colors seems to be too weak whenever a fragile item is not contained in a bin (implementation of (4)). The problem can be fixed by replacing the implication by an equivalence (`FD.equi`). The correct (4) in the constraint model is $\forall n : i_{fragile} \in s_n \leftrightarrow color(s_n) = red$. That means that the implementation was correct

but the constraint model had a flaw. After applying the fix the solver produces a proper solution.

15.4.2 Approaches for Dealing with Realistic Applications

Realistic problems may have thousands of propagators and variables. It is impossible and without any practical use to represent all at once. This section proposes techniques for selecting problem-relevant fractions of propagators or variables. This scheme allows for a user-controlled incremental exploration of the graphs which is essential for the investigation of large problems.

A common approach of designing a constraint model is to decompose the problem into subproblems and to decompose these subproblems until predefined propagators can be used. Since procedures implement subproblems, it seems reasonable to structure propagators, sub-procedures, and variables according to the procedures which stated them. This requires the introduction of procedure nodes to the graph views. A procedure node is depicted as circle.

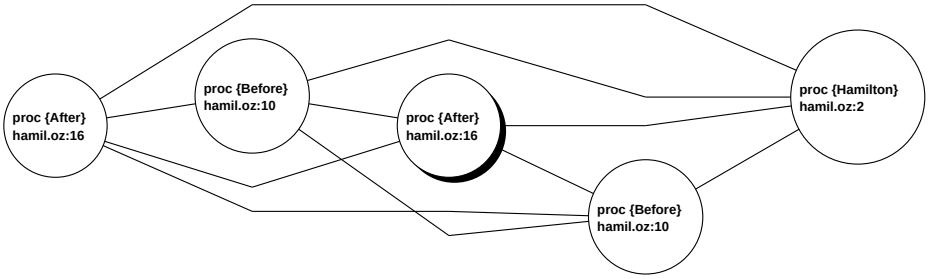
Selection via the Tree of Execution Traces The tree representation of a constraint program's execution trace is used to select propagators and variables. A node represents a procedure execution imposing propagators. By clicking on a node, a possible action is to select the propagators created by the corresponding procedure invocation. Incremental expansion of the tree makes possible to handle large collections of propagators and variables. Different selection schemes, *e.g.*, all propagators stated by a procedure with respectively without their sub-procedures, extend the functionality.

Collapsing and Expanding Propagator and Procedure Nodes A common technique for handling large collections of data represented by graphs is to collapse and expand appropriate subsets of nodes to single nodes. The propagator graph view is proposed to be partitioned into subsets of nodes according to the procedures which created corresponding propagators. That means a collapsed node represents a collection of propagators and sub-procedures. This is very close to the model the programmer has in mind when structuring the problem and hence, is very intuitive.

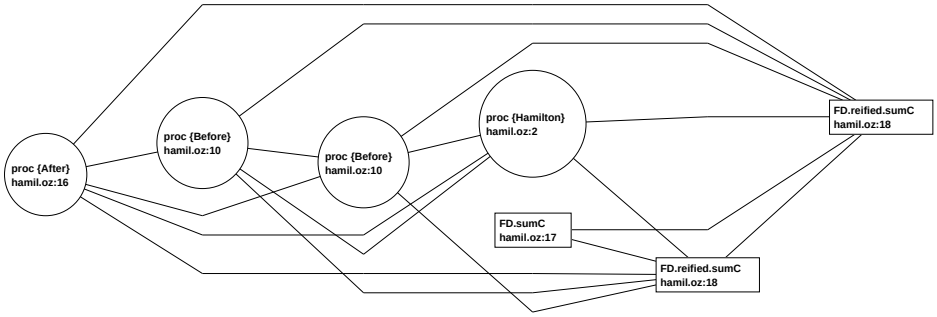
A procedure node represents a collection of propagator nodes and sub-procedure nodes. It takes as its parameters the union of the parameters of all represented propagators and sub-procedures.

Figure 15.10 shows the expansion of the marked procedure node to a collection of propagator nodes. Expansion can be undone by collapsing propagator and procedure nodes to a single procedure node.

Filtering Propagators and Variables Another interesting feature is the option of displaying only those propagators resp. variables which meet a criterion specified by the user. For example, it might be interesting to limit the investigation to those propagators that are connected to boolean variables when symptoms of a bug suggest that.



(a) Fully collapsed procedure graph, *i.e.*, all propagator nodes are collapsed.



(b) Partially collapsed propagator graph, *i.e.*, a procedure's node is expanded to its propagator nodes.

Figure 15.10: Transition of a graph view by expanding a procedure node.

15.4.3 Additional Features

This section discusses features of the Investigator not covered before but important for effective use of the tool.

Navigating Through Graphs Navigation through the different graph views is done by menus associated with nodes and edges of the respective views. Figure 15.11 shows possible transitions from one view to another one. A history mechanism is also available, allowing to recall previous views by moving in the chain of views produced so far.

To further improve navigation and to keep track of a certain node in different views, the Investigator is able to mark nodes in graph views which then remain marked throughout all views. Additionally, the Investigator automatically marks nodes of variables with which the session was initiated (Figure 15.7) and in case there is a failed propagator, the node of this propagator (Figure 15.5).

Changing the Representation of Nodes The Investigator provides a plug-in mechanism for changing the representation of variables and propagators. This enables the user to produce a more obvious and intuitive representation. For example, a propagator for a

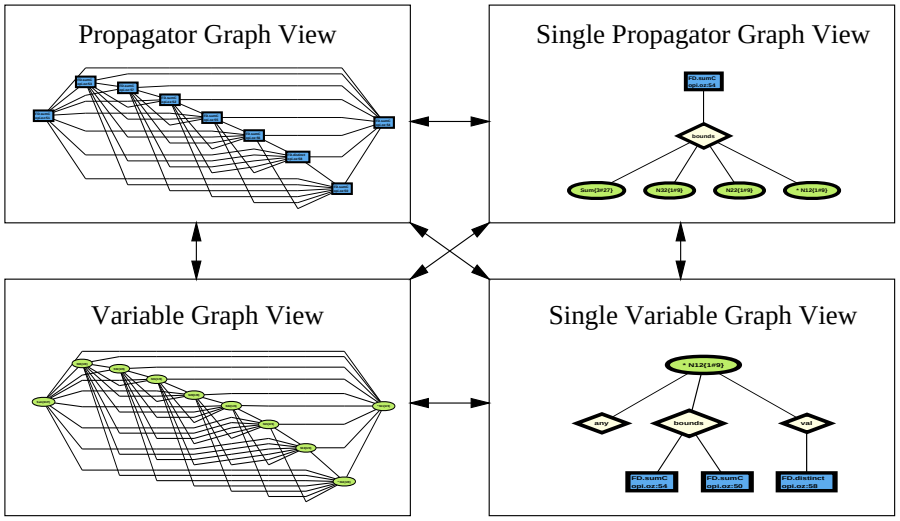


Figure 15.11: Navigation overview.

constraint in a scheduling application might be represented as Gantt-chart, reflecting its role in the concrete application.

15.5 Implementation

The Investigator is implemented in Mozart Oz and uses the graph visualization system *daVinci* [36]. The implementation consists of two parts: first, the reflection of variables and propagators in the solver into Oz data structures, and second, the construction of the graphs and the generation of the corresponding *daVinci* terms.

Reflection The reflection part uses `{Constraint.reflectSpace Rs Vs Cs}` (see Section 14.2) which takes a list of variables `Rs` and returns a list denoting the set of propagators `Cs` reachable from `Rs` and a list denoting the set of variables `Vs` occuring as parameters in `Cs`. The elements of `Vs` are reflected variables and the elements of `Cs` are reflected propagators. A reflected variable stores the name, the basic constraint, the references to the propagators that are waiting for the variable's basic constraint to change, and the actual variable. A reflected propagator stores the propagator's name and parameters and a reference to the actual propagator.

The list `Vs` and `Cs` are translated into compact representations for the connections between propagators and variables using sets of integers. Each propagator and variable is assigned a unique integer. A reflected propagator stores its parameters as an integer set, also a reflected variable stores the propagators for which it is a parameter as an integer set. The resulting data structures make explicit the connections between propagators and variables, *i.e.*, which are variables acting as parameters of propagators and which propagators are waiting for a variable's basic constraint to change.

Graph Generation The edges and nodes of the individual graph views are computed by

the set operations described in Section 15.3 and are further translated to *daVinci* terms. The *daVinci* terms are augmented by menus to allow for comfortable user interaction.

The complexity of the graph-generation algorithm is quadratic in the worst case and depends in practice on the degree of connectivity of the constraint network, *i.e.*, if the propagators can be stated in reasonable time then the corresponding graph can be computed in reasonable time too.

Collapsing and expanding of procedures and propagators required extra data structures for procedures. These are an extension of the data structures used for propagators and contain additionally references to sub-procedures and propagators stated by the respective procedure. Due to the set-based implementation, the changes can be factorized out nicely.

15.6 Related Work and Discussion

The tools discussed in this section focus on improving performance. Since the presented approach is orthogonal, it can be used to supplement existing tools.

The GRACE constraint debugger by Meier [91] supplements the Prolog-based constraint programming system ECL^iPS^e [76] and is intended to support performance debugging of finite domain constraint programming. The constraint program has to be appropriately instrumented to be run under GRACE. The debugging model of GRACE is based upon the Prolog-box-model. It is able to follow individual propagation steps in the trace and to inspect the backtrack stack of finite domain variables. Furthermore, GRACE is highly configurable by assigning user-written code to each propagation step.

The Oz Explorer is a graphical search engine which visualizes the search tree as search proceeds. It allows the user to control search, *e.g.* one can interrupt search and can resume search from a branch different from the branch explored last. The Explorer is extendable by plug-ins, to provide different views of nodes in the search tree. The Explorer is particularly useful for optimizing search heuristics according to the topology of the search tree. In conjunction with the Investigator, debugging performance and correctness issues of constraint programs is actively supported.

The search tree debugger of CHIP [134, 40] is largely influenced by the Oz Explorer. Its focus is performance debugging. It provides different types of views, mostly in a compact matrix-like fashion, to provide the user with more detailed information about search and constraint propagation. A nice feature is to analyze the evolution of constraints and variables along a search path. This is certainly most valuable for optimizing search heuristics.

Section 15.4.2 points out the difficulties of presenting a large number of propagators to the user no matter what kind of representation is used. The idea to represent a set of propagators by a single propagator is quite natural. Goulard and Benhamou follow this idea too and propose so-called S-boxes [56]. An S-box represents a set of individual propagators and can be interactively defined during a debugger session. An S-box appears like a single propagator, *i.e.*, it has a number of parameters and implements a certain constraint. An S-box can create another S-box and so on leading to a hierarchy

of S-boxes. This hierarchy is similar to the tree of execution traces but the way an S-box can be defined is more flexible. Thus, S-boxes are an ideal supplement to the selection of propagators by a tree of execution traces. The Investigator can benefit from using S-boxes as another alternative way to handle large numbers of propagators.

Chapter 16

Conclusion

This chapter summarizes the contributions presented in this dissertation (Section 16.1) and proposes future work extending and continuing the presented research (Section 16.2).

16.1 Contributions

Constraint Propagation Engines

Model An architecture for propagator-based constraint solver including an interface for separating propagation services and domain solver is developed. The interface separates filter algorithms from propagators to make host solver independent filter design and implementation possible.

Integration and Implementation Propagation services for the Mozart virtual machine are designed and implemented. The implementation is completely orthogonal to the implementation of the rest of the virtual machine. A high-level C++-constraint programming interface CPI for implementing domain solvers for Mozart is designed and implemented. The interface supports host system-independent implementation of filters.

Combining the propagation services with a finite domain solver implemented by the CPI results in a finite domain propagation engine being more efficient *w.r.t.* to plain propagation performance than today's best commercial systems. The implemented propagation engines meet the standards of production-quality systems and are successfully used in industrial applications.

Performance Evaluation The propagation performance of the propagation engines of various solvers with Mozart by using a configurable inconsistent constraint is compared and evaluated. Further, the impact of various interfaces on the propagation performance of Mozart is analyzed in detail.

Finite Integer Set Constraints

Filter Generation A scheme for generating filter algorithms for finite integer set constraints performing bounds and cardinality reasoning is developed and implemented.

A Finite Integer Set Solver for Mozart An expressive, efficient, production-quality finite integer set constraint domain solver for Mozart is designed and implemented. Set constraints taking the integer domain into account are introduced. This leads to new techniques for breaking symmetries. The new functionality opens finite set constraint programming for computer linguistic applications apart from combinatorial optimization problems.

First-class Constraints

Constraints as First-class Citizens Constraints are introduced as first-class citizens to constraint programming. Novel constraint programming techniques, as early failure detection, constraint optimization, and explicit garbage collection are developed, to demonstrate the benefits of using first-class constraints. First-class constraints are prototypically integrated in Mozart as a natural and orthogonal extension of the architecture presented in Part I.

Debugging Propagator-based Solvers A novel debugging scheme based on graph views for propagator-based constraint solvers is developed and a usable prototype of an interactive debugging tool is implemented. This tool is based on graph views and is implemented using first-class constraints. The debugging tool is not restricted to any constraint domain, features intuitive data presentation and user interaction. Further, different approaches for handling constraint engines with a large number of propagators and constraint variables are proposed.

16.2 Future Work

Scheduling Strategies for Propagators The strategies for scheduling propagators are not yet very well investigated. Laburthe proposes in [83] a scheduling heuristics which assigns propagators priorities according to their computational complexity and the events they are waiting for. Mozart provides more restricted control by two sets of runnable propagators. Other schemes are thinkable and open for investigation and analysis.

Libraries The presented implementation and interface models for propagation engines including their integration into Mozart as described in Part I have proved to be very successful. Using the developed concepts to build a constraint library is a demanding task. Such a library is free to drop obstacles as variable equality represented in the constraint store to further improve performance and to simplify the implementation. The development of the constraint library FIGARO [64] is an instance of such a project.

Formal Justification of the Filter Generation Scheme Propagators using filters generated by the scheme presented in Chapter 12 provide evidence that the scheme is useful.

A formal justification of the presented scheme would guarantee provable properties and is still open.

Filter Generation for Generic Finite Set Constraints The scheme to generate filter algorithms for finite set operators is not able to generate filters for generic set operators where the number of parameters is not fixed. Extending the presented scheme in this direction would be extremely useful.

Filters Generation for Other Constraint Domains It would be interesting to investigate the application of the filter generation scheme to other constraint domains than finite integer sets. For example, the generation of filter algorithms for arithmetic constraints over a numerical domain (*e.g.*, finite domains ad reals) seems promising.

First-class Constraints Part III promotes constraints to first-class citizens and develops novel programming techniques on top of them. I am convinced that there are more promising techniques and application areas to be explored. For example, taking advantage of first-class constraints in search strategies appears to be promising (see [93, 94] for applications in proof-planning). The work of Hong on RISC-CLP(Real) [68] and RISC-CLP(CF) [69] opens a promising field for combining symbolic and domain reasoning which seems to be suitable for being tackled with first-class constraints.

Hybrid Solvers A hybrid solver consists of various sub-solver with different strengthens and weaknesses. It is desirable to automatically submit constraints to the most appropriate solver or to combine solvers depending on the currently present constraints in a solver. First-class constraints provide the right expressiveness to build such solvers and open an existing direction of research.

Constraint Debugging The combination of the constraint debugging scheme based on graph views presented in Chapter 15 with the concept of S-boxes [56] is worth investigating to improve the capabilities to handle large applications. Further, automatic analysis of erroneous solvers (à la the computation of smallest inconsistent sets with first-class constraints in Section 14.3.4) might lead to novel debugging schemes and tools.

Appendix A

Performance Figures Summary

m×n	Mozart Oz 1.2.0	*ECL^{i}PS^e* 5.2		SICStus 3.8.5		GNU Prolog 1.2.1		ILOG Solver 5.0	
1×1	(2.47%)	8.35	(0.07%)	6.67	(0.78%)	1.61	(0.21%)	2.17	(0.08%)
1×10	(2.60%)	8.66	(0.09%)	2.81	(0.15%)	2.32^{-1}	(0.22%)	2.15	(0.03%)
1×100	(2.85%)	8.87	(0.18%)	2.38	(0.20%)	3.67^{-1}	(0.48%)	2.14	(3.26%)
1×1000	(2.24%)	9.45	(0.48%)	2.72	(2.32%)	2.95^{-1}	(10.27%)	2.23	(1.80%)
1×10000	(1.35%)	35.53	(1.17%)	11.92	(1.78%)	1.33	(0.81%)	4.90	(0.95%)
1×100000	(0.77%)	–		–		core dump		244.27	(0.88%)
10×1	(2.37%)	–		–		2.94^{-1}	(0.34%)	1.64	(0.11%)
10×10	(2.77%)	–		–		3.70^{-1}	(0.53%)	2.05	(0.46%)
10×100	(1.42%)	–		–		2.72^{-1}	(11.51%)	2.47	(3.03%)
10×1000	(2.55%)	–		–		1.25^{-1}	(0.40%)	2.83	(0.71%)
10×10000	(1.16%)	–		–		core dump		26.57	(1.32%)
100×1	(3.26%)	–		–		3.31^{-1}	(7.72%)	1.80	(5.33%)
100×10	(2.93%)	–		–		2.55^{-1}	(9.38%)	2.72	(4.27%)
100×100	(1.65%)	–		–		1.19^{-1}	(0.52%)	3.08	(0.76%)
100×1000	(0.50%)	–		–		core dump		4.89	(0.15%)
1000×1	(3.33%)	–		–		1.55^{-1}	(4.50%)	3.00	(1.93%)
1000×10	(2.06%)	–		–		1.11^{-1}	(0.74%)	3.96	(1.18%)
1000×100	(0.62%)	–		–		core dump		4.61	(0.31%)
10000×1	(1.45%)	–		–		1.06^{-1}	(0.41%)	2.99	(0.52%)
10000×10	(1.84%)	–		–		core dump		4.16	(0.50%)
100000×1	(2.09%)	–		–		core dump		2.94	(0.58%)

Table A.1: This table contains the speedup figures and the corresponding coefficients of variation for all benchmarked systems. The row for Mozart Oz 1.2.0 contains only the coefficients of variation since the speedup is of course always 1. An entry '–' means the benchmark was not done due to too significant variation of the measured results. An entry 'core dump' means that the program terminated by dumping a core file, *i.e.*, the program crashed.

Bibliography

[1] Emile Aarts and Jan Karel Lenstra. *Local Search in Combinatorial Optimization*. John Wiley, 1997.

[2] Abderrahamane Aggoun and Nicolas Beldiceanu. Time Stamps Techniques for the Trailed Data in Constraint Logic Programming Systems. In S. Bourgault and M. Dincbas, editors, *Actes du Séminaire 1990 de programmation en Logique*, pages 487–509, Trégastel, France, May 1990. CNET, Lannion, France.

[3] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. Logic Programming Series. The MIT Press, Cambridge, MA, USA, 1991.

[4] Yasuhiro Ajiro and Kazunori Ueda. Kima – an automated error correction system for concurrent logic programs. In Mireille Ducassé, editor, *Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG 2000)*, August 2000. Available at `http://www.irisa.fr/lande/ducasse/aadebug2000/proceedings.html`.

[5] Yasuhiro Ajiro, Kazunori Ueda, and Kenta Cho. Error-correcting source code. In Michael Maher and Jean-François Puget, editors, *Proceedings of the Forth International Conference on Principles and Practice of Constraint Programming*, volume 1520 of *Lecture Notes in Computer Science*, pages 40–54, Pisa, Italy, October 1998. Springer-Verlag.

[6] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *Operations Research Society of America, Journal on Computing*, 3(2):149–156, 1991.

[7] Krzysztof R. Apt. The role of commutativity in constraint propagation algorithms. *ACM Transactions on Programming Languages and Systems*, 22(6):1002–1036, November 2000.

[8] Francisco Azevedo and Pedro Barahona. Modelling digital circuits problems with set constraints. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Proceedings of the First International Conference on Computational Logic – CL2000*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 414–428, London, UK, July 2000. Springer Verlag.

[9] C. Beeri, S. Nagvi, O. Shmueli, and S. Tsur. Set constructors in a logic database language. *Journal of Logic Programming*, 10(3):181–232, 1991.

[10] Nicolas Beldiceanu. An example of introduction of global constraints in CHIP: Application to block theory problems. Technical Report TR-LP-49, ECRC, Munich, Germany, May 1990.

[11] Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1994.

[12] Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 32(1):1–24, 1997.

[13] Henri Beringer and Bruno de Backer. Combinatorial problem solving in constraint logic programming with cooperating solvers. In C. Beierle and L. Plümer, editors, *Logic programming: Formal methods and practical applications*, pages 245–272. Elsevier, 1995.

[14] Christian Bessière and Marie-Odile Cordier. Arc-consistency and arc-consistency again. In *AAAI-93: Proceedings 11th National Conference on Artificial Intelligence*, Washington, DC, 1993.

[15] Christian Bessière, Eugene Freuder, and Jean-Charles Régin. Using inference to reduce arc consistency computation. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 592–598, Montréal, Québec, Canada, 1995.

[16] Stefano Bistarelli, Helene Fargier, Ugo Montanari, Francesca Rossi, Thomas Schiex, and Gerard Verfaillie. Semiring-based csps and valued csps: Frameworks, properties, and comparison. *Constraints*, 4(3), September 1999.

[17] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Constraint solving over semirings. In Chris S. Mellish, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 232–238, Montréal, Québec, Canada, August 1995. Morgan Kaufmann Publishers, San Mateo, CA.

[18] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint solving and optimization. *Journal of the ACM*, 44(2):201–236, March 1997.

[19] P. Bruscoli, A. Dovier, E. Pontelli, and G. Rossi. Compiling intensional sets in CLP. In *Proceedings of the International Conference on Logic Programming*, pages 647–661. MIT Press, January 1994.

[20] Björn Carlson. *Compiling and Executing Finite Domain Constraints*. Dissertation, Computing Science Department, Uppsala University, and SICS – Swedish Institute of Computer Science, Box 311 S-751 05 Uppsala, Sweden, 1995. Uppsala Theses in Computing Science 21, and SICS Dissertation Series 18.

[21] Björn Carlson and Mats Carlsson. Compiling and executing disjunctions of finite domain constraints. In *ICLP'95, International Conference on Logic Programming*, MIT Press Series in Logic Programming, Kanagawa, Japan, 1995. The MIT Press.

[22] Björn Carlson, Mats Carlsson, and Sverker Janson. The implementation of AKL(FD). In *Proceedings of the International Logic Programming Symposium*, pages 227–241, Portland, OR, USA, 1995. The MIT Press, Cambridge.

[23] Mats Carlsson. Personal communication, February 2001.

[24] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206, Southampton, UK, 1997. Springer.

[25] Manuel Carro and Manuel Hermenegildo. Some design issues in the visualization of constraint logic program execution. In *AGP 1998, Joint Conference on Declarative Programming*, Corunna, Spain, July 1998.

[26] Yves Caseau, François-Xavier Josset, and François Laburthe. CLAIRE: Combining sets, search and rules to better express algorithms. In Danny De Schreye, editor, *Proceedings of the 1999 International Conference on Logic Programming*, pages 245–259, Las Cruces, NM, USA, November 1999. The MIT Press.

[27] Yves Caseau and François Laburthe. Improved CLP scheduling with task intervals. In *Proceedings of the International Conference on Logic Programming*, pages 369–383, 1994.

[28] Yves Caseau and François Laburthe. Cumulative scheduling with task intervals. In *Joint International Conference and Symposium on Logic Programming*, 1996.

[29] Yves Caseau and François Laburthe. Solving various weighted matching problems with constraints. In Gert Smolka, editor, *Principles and Practice of Constraint Programming—CP97, Proceedings of the Third International Conference*, Lecture Notes in Computer Science 1330, pages 17–31, Schloss Hagenberg, Linz, Austria, October/November 1997. Springer-Verlag, Berlin.

[30] Kenta Cho and Kazunori Ueda. Diagnosing non-well-moded concurrent logic programs. In Michael Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 251–229, Bonn, Germany, 1996. The MIT Press, Cambridge.

[31] Vašek Chvátal. *Linear Programming*. W.H. Freeman and Company, 41 Madison Avenue, New York 10010, 1983.

[32] Philippe Codognet and Daniel Diaz. Compiling constraints in `clp(FD)`. *Journal of Logic Programming*, 27(3):185–226, June 1996.

[33] Alain Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, pages 85–99, 1984.

[34] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge;London, 1990.

[35] George B. Dantzig. *Linear Programming and Extensions*. Princeton Landmarks in Mathematics and Physics. Princeton University Press, Princeton, NJ, 1963.

[36] The graph visualization system *daVinci*. `http://www.informatik.uni-bremen.de/~davinci/`, 2000.

[37] Pierre Deransart, Manuel V. Hermenegildo, and Jan Małuszyński, editors. *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging*, volume 1870 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 2000.

[38] Daniel Diaz. *GNU Prolog 1.2.1 – A Native Prolog Compiler with Constraint Solving over Finite Domains*. `http://www.gnu.org/software/prolog`, July 2000.

[39] Daniel Diaz and Philippe Codognet. The GNU prolog systems and its implementation. In *ACM Symposium on Applied Computing*, Como, Italy, 2000. Documentation and system available at `http://www.gnu.org/software/prolog`.

[40] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, Japan, December 1988. Institute for New Generation Computer Technology (ICOT),Tokyo, Japan.

[41] DiSCiPl. Debugging systems for constraint programming. `http://discipl.inria.fr/`, 1999.

[42] Agostino Dovier, Enrico Pontelli, Carla Piazza, and Gianfranco Rossi. Sets and constraint logic programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861 – 931, September 2000.

[43] Agostino Dovier and Gianfranco Rossi. Embedding Extensional Finite Sets in CLP. In *Proceedings of the International Logic Programming Symposium*, pages 540–556, Vancouver, Canada, 1993.

[44] Denys Duchier. Lexicalized syntax and topology for non-projective dependency grammar. Submitted, April 2001. Available at `http://www.ps.uni-sb.de/Papers/abstracts/duchier-fgmol2001.html`.

[45] Denys Duchier and Claire Gardent. A constraint-based treatment of descriptions. In H.C. Bunt and E.G.C. Thijsse, editors, *Third International Workshop on Computational Semantics (IWCS-3)*, pages 71–85, Tilburg, NL, January 1999.

[46] Denys Duchier, Leif Kornstaedt, Tobias Müller, Christian Schulte, and Peter Van Roy. *System Modules*. The Mozart Consortium – Mozart Oz 1.2.0, May 2001. Available at `http://www.mozart-oz.org/documentation/system/index.html`.

[47] Denys Duchier, Leif Kornstaedt, and Christian Schulte. *The Oz Base Environment*. The Mozart Consortium – Mozart Oz 1.2.0, May 2001. Available at `http://www.mozart-oz.org/documentation/base/index.html`.

[48] Denys Duchier and Joachim Niehren. Dominance constraints with set operators. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Proceedings of the First International Conference on Computational Logic (CL2000)*, volume 1861 of *LNCS*, pages 326–341, London, UK, July 2000. Springer.

[49] Jeff Foster. CLP(SC): Implementation and efficiency considerations. In *Proceedings of the Set-Constraint Workshop at the 1996 CognitivePsychology*, Boston, Massachusetts, 1996.

[50] M. Fröhlich. *Inkrementelles Graphlayout im Visualisierungssystem daVinci*. PhD thesis, Universität Bremen, Fachbreich 3 – Mathematik und Informatik, November 1997. In German.

[51] Thom Frühwirth. Theory and practice of constraint handling rules. *Special Issue on Constraint Logic Programming, Journal of Logic Programming*, 37(1–3), October 1998.

[52] Thom Frühwirth and Slim Abdennadher. *Constraint–Programmierung: Grundlagen und Anwendungen*. Springer-Verlag, Berlin, Germany, 1997.

[53] Yan Georget and Philippe Codognet. Encoding global constraints in semiring-based constraint solving. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI'98)*, Taipeh, Taiwan, October 1998. IEEE Press.

[54] Carmen Gervet. *Set Intervals in Constraint-Logic Programming: Definition and Implementation of a Language*. PhD thesis, Université de France-Compté, September 1995. European Thesis.

[55] Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

[56] Frédéric Goualard and Frédéric Benhamou. Debugging constraint programs by store inspection. In Deransart et al. [37], pages 273–297.

[57] Martin Grötschel, Michael Jünger, and Gerhard Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research Society of America, Operations Research*, 32(6):1195–1220, 1984.

[58] J. Gu. Efficient local search for very large-scale satisifiability problems. *SIGART Bulletin*, 3(1):8–12, 1992.

[59] David R. Hanson. *C Interfaces and Implementations – Techniques for Creating Reusable Software*. Professional Computing Series. Addison-Wesley Publishing Company, 1996.

[60] Seif Haridi and Nils Franzén. *Tutorial of Oz*. The Mozart Consortium – Mozart Oz 1.1.1, February 2000. Available at `http://www.mozart-oz.org/documentation/tutorial/index.html`.

[61] Warwick Harvey and Peter J. Stuckey. Constraint representation for propagation. In M. Maher and J.-F. Puget, editors, *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming (CP98)*, Lecture Notes in Computer Science, pages 235–249, Pisa, Italy, October 1998. Springer-Verlag.

[62] Nevin Heintze, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. Meta-programming in CLP($\mathcal{R}$). *Journal of Logic Programming*, 33(3):221–259, December 1997.

[63] Martin Henz. *Objects for Concurrent Constraint Programming*, volume 426 of *International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, MA, USA, October 1997.

[64] Martin Henz, Tobias Müller, and Ka Boon Ng. Figaro: Yet another constraint programming library. In I. de Castro Dutra, V. Santos Costa, G. Gupta, E. Pontelli, M. Carro, and P. Kacsuk, editors, *Proceedings of the Workshop on Parallelism and Implementation Technology for Constraint Logic Programming*, pages 86–96, New Mexico State University, Las Cruces, New Mexico, December 1999.

[65] Karla L. Hoffman and Manfred Padberg. Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39(6):657 – 682, 1993.

[66] Karla L. Hoffman, Manfred Padberg, and Russell A. Rushmeier. Recent advances in exact optimization of airline scheduling problems, July 1995.

[67] Christian Holzbaur. *Specification of Constraint Based Inference Mechanisms through Extended Unification*. PhD thesis, Technisch-Naturwissenschaftliche Fakultät der Technischen Universität Wien, October 1990.

[68] Hoon Hong. RISC-CLP(Real): Constraint logic programming over real numbers. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*. MIT Press, 1993.

[69] Hoon Hong. RISC-CLP(CF): Constraint logic programming over functions. In *Logic Programming and Automated Reasoning (LPAR'94)*, July 1994.

[70] ILOG S.A., `http://www.cplex.com/`. *Using the CPLEX Callable Library and Base System, Version 5.0*, 1997.

[71] ILOG S.A., `http://www.ilog.com/`. *ILOG Concert Technology 1.0, User's Manual*, August 2000.

[72] ILOG S.A., `http://www.ilog.com/`. *ILOG Scheduler 5.0, User's Manual*, July 2000.

[73] ILOG S.A., `http://www.ilog.com/`. *ILOG Solver 5.0, User's Manual*, August 2000.

[74] Intelligent Systems Laboratory. *SICStus Prolog User's Manual*. SICS Research Report, Swedish Institute of Computer Science, `http://www.sics.se/isl/sicstus.html`, 2000.

[75] International Computers Limited and IC-Parc. *ECL^{i}PS^e, Library Manual, Release 5.0*, November 2000.

[76] International Computers Limited and IC-Parc. *ECL^{i}PS^e, User Manual, Release 5.0*, November 2000.

[77] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, January 1987. ACM.

[78] Joxan Jaffar and Michael M. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–582, May 1994. Special Issue: Ten Years of Logic Programming.

[79] Raj Jain. *The Art of Computer Systems Performance Analysis – Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, Inc., 1991.

[80] Alexander Koller and Joachim Niehren. Constraint programming in computational linguistics. In D. Barker-Plummer, D. Beaver, J. van Benthem, and P. Scotto di Luzio, editors, *Proceedings of the eight CSLI Workshop on Logic Language and Computation*. CSLI Press, 2000.

[81] Dexter Kozen. Set constraints and logic programming. In Jean-Pierre Jouannaud, editor, *Proceedings of the 1st International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, München, 1994. Springer Verlag.

[82] Gabriel Kuper. *Logic Programming with Sets*. Academic Press, New York, N.Y., 1990.

[83] François Laburthe. CHOCO: implementing a CP kernel. In Nicolas Beldiceanu, Warwick Harvey, Martin Henz, François Laburthe, Eric Monfroy, Tobias Müller, Laurent Perron, and Christian Schulte, editors, *Proceedings of the Workshop on Techniques for Implementing Constraint Programming Systems - TRICS*, pages 71–85, Singapore, September 2000.

[84] B. Legeard and E. Legros. Short Overview of the CLPS System. In J. Małuszyński and M. Wirsing, editors, *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 431–433, Passau, Germany, August 1991. Springer Verlag.

[85] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[86] Suresh Manandhar. An attributive logic of set descriptions and set operations. In *Proceedings of the Annual Meeting of the Association of Computational Linguistics*, 1994.

[87] Kim Marriott and Peter J. Stuckey. *Programming with Constraints. An Introduction*. The MIT Press, Cambridge, MA, USA, 1998.

[88] Michael Mehl. *The Oz Virtual Machine - Records, Transients, and Deep Guards*. PhD thesis, Technische Fakultät der Universität des Saarlandes, 1999.

[89] Michael Mehl, Ralf Scheidhauer, and Christian Schulte. An Abstract Machine for Oz. In M. Hermenegildo and S. D. Swierstra, editors, *Programming Languages: Implementations, Logics and Programs, 7th International Symposium, PLILP'95*, volume 982 of *Lecture Notes in Computer Science*, pages 151–168, Utrecht, The Netherlands, 20–22 September 1995. Springer-Verlag, Berlin-Heidelberg.

[90] Kurt Mehlhorn and Sven Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In Rina Dechter, editor, *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming – CP 2000*, volume 1984 of *Lecture Notes in Computer Science*, pages 306–319, Singapore, September 2000. Springer Verlag.

[91] Micha Meier. Debugging constraint programs. In Ugo Montanari and Francesca Rossi, editors, *Proceedings of the First International Conference on Principles*

and Practice of Constraint Programming, volume 976 of *Lecture Notes in Computer Science*, pages 204–221, Cassis, France, September 1995. Springer Verlag.

[92] Micha Meier and Pascal Brisset. Open architecture for CLP. Technical Report ECRC-95-10, European Computer-Industry Research Centre, European Computer-Industry Research Center GmbH, Arabellastrasse 17, D-81925 Munich, 1995.

[93] Erica Melis, Jürgen Zimmer, and Tobias Müller. Extensions of constraint solving for proof planning. In Werner Horn, editor, *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 229–233, Berlin, August 2000. IOS Press.

[94] Erica Melis, Jürgen Zimmer, and Tobias Müller. Integrating constraint solving into proof planning. In Hélène Kirchner and Christophe Ringeissen, editors, *Frontiers of Combining Systems – Third International Workshop, FroCos 2000*, volume 1794 of *Lecture Notes in Artificial Intelligence*, pages 32–46, Nancy, France, March 2000. Springer Verlag.

[95] Ugo Montanari. Networks of constraints: fundamental properties and application to picture processing. *Information Sciences*, 7:95–132, 1974.

[96] Ugo Montanari and Francesca Rossi. True concurrency in concurrent constraint programming. In Vijay Saraswat and Kazunori Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 694–713, San Diego, USA, June 1991. The MIT Press.

[97] Johan Montelius. *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages*. PhD thesis, SICS Swedish Institute of Computer Science, SICS Box 1263, S-164 28 Kista, Sweden, April 1997. SICS Dissertation Series 25.

[98] David S. Moore and George P. McCabe. *Introduction to the Practice of Statistics*. W. H. Freeman and Company, 1999.

[99] Mozart Consortium. The Mozart Programming System 1.2.0. Documentation and system available from `http://www.mozart-oz.org`, Programming Systems Lab, Saarbrücken, Swedish Institute of Computer Science, Stockholm, and Université catholique de Louvain, February 2001.

[100] Tobias Müller. Solving set partitioning problems with constraint programming. In *Proceedings of the Sixth International Conference on the Practical Application of Prolog and the Forth International Conference on the Practical Application of Constraint Technology – PAPPACT98*, pages 313–332, London, UK, March 1998. The Practical Application Company Ltd.

[101] Tobias Müller. Practical investigation of constraints with graph views. In Rina Dechter, editor, *Proceedings of the Sixth International Conference on Principles*

and Practice of Constraint Programming – CP 2000, volume 1984 of *Lecture Notes in Computer Science*, pages 320–336, Singapore, September 2000. Springer Verlag.

[102] Tobias Müller. Promoting constraints to first-class status. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Proceedings of the First International Conference on Computational Logic – CL2000*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 429–447, London, UK, July 2000. Springer Verlag.

[103] Tobias Müller. *The Mozart Constraint Extensions Reference*. The Mozart Consortium – Mozart Oz 1.2.0, May 2001. Available at `http://www.mozart-oz.org/documentation/cpiref/index.html`.

[104] Tobias Müller. *The Mozart Constraint Extensions Tutorial*. The Mozart Consortium – Mozart Oz 1.2.0, May 2001. Available at `http://www.mozart-oz.org/documentation/cpitut/index.html`.

[105] Tobias Müller. Supplementary material to the thesis *Constraint Propagation in Mozart*, 2001. Available at `http://www.ps.uni-sb.de/~tmueller/thesis/`.

[106] Tobias Müller and Martin Müller. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *13. Workshop Logische Programmierung*, pages 104–115, Technische Universität München, 17–19 September 1997.

[107] Tobias Müller and Jörg Würtz. Extending a concurrent constraint language by propagators. In Jan Małuszyński, editor, *Proceedings of the International Logic Programming Symposium*, pages 149–163, Long Island, NY, USA, 1997. The MIT Press, Cambridge.

[108] Tobias Müller and Jörg Würtz. Embedding propagators in a concurrent constraint language. *The Journal of Functional and Logic Programming*, 1999(Special Issue 1):Article 8, April 1999. Published on the Internet: `http://mitpress.mit.edu/JFLP/`, ISSN 1080–5230, MIT Press Journals, Five Cambridge Center, Cambridge, USA.

[109] Robert B. Murray. *C++ Strategies and Tactics*. Professional Computing Series. Addison-Wesley Publishing Company, 1993.

[110] George L. Nemhauser and Laurence A. Wolsey. *Integer and combinatorial optimization*. John Wiley and Sons, 1988.

[111] Ka Boon Ng, Chiu Wo Choi, Martin Henz, and Tobias Müller. GIFT: a generic interface for reusing filtering algorithms. In Nicolas Beldiceanu, Warwick Harvey,

Martin Henz, François Laburthe, Eric Monfroy, Tobias Müller, Laurent Perron, and Christian Schulte, editors, *Proceedings of the Workshop on Techniques for Implementing Constraint Programming Systems - TRICS*, pages 86–100, Singapore, September 2000.

[112] William J. Older and Frédéric Benhamou. Programming in CLP(BNR). In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming*, pages 239–249, Bell Northern Research, Computing Research Laboratory, P.O. Box 3511, Station C KIY 4H7 Ottawa, Ontario, Canada, April 1993. Reference materials for workshop participants only, Organized by Brown University.

[113] Leszek Pacholski and Andreas Podelski. Set constraints: A pearl in research on constraints. In Gert Smolka, editor, *Principles and Practice of Constraint Programming—CP97, Proceedings of the Third International Conference*, volume 1330 of *Lecture Notes in Computer Science*, pages 549–561. Springer Verlag, 1997.

[114] Jean-François Puget. PECOS: A high level constraint programming language. In *Proceedings of the First Singapore International Conference on Intelligent Systems (SPICIS)*, pages 137–142, Singapore, September/October 1992.

[115] Jean-François Puget. Finite Set Intervals. In *Proceedings Workshop on Set Constraints, held in Conjunction with CP'96*, Boston, Massachusetts, 1996.

[116] Jean-François Puget and Michel Leconte. Beyond the glass box: Constraints as objects. In John Lloyd, editor, *Logic Programming – Proceedings of the 1995 International Symposium*, pages 513–527, Portland, OR, USA, December 1995. The MIT Press, Cambridge.

[117] Jean-François Puget. A C++ implementation of CLP. In *Proceedings of the Second Singapore International Conference on Intelligent Systems (SPICIS)*, pages 256–261, Singapore, November 1994.

[118] Jean-François Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 359–366, Madison, WI, USA, July 1998. AAAI Press/The MIT Press.

[119] Jean-Charles Régin. Personal communication, April 2001.

[120] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 362–367, Seattle, WA, USA, 1994. AAAI Press.

[121] Robert Rodošek and Mark Wallace. A generic model and hybrid algorithm for hoist scheduling problems. In M. Maher and J.-F. Puget, editors, *Proceedings*

of the Fourth International Conference on Principles and Practice of Constraint Programming (CP98), Lecture Notes in Computer Science, pages 385–399, Pisa, Italy, October 1998. Springer-Verlag.

[122] Robert Rodošek, Mark G. Wallace, and Mozafar T. Haijan. A new approach to integrate mixed integer programming with CLP. In *Proceedings of the Workshop on Constraint Programming Applications, in conjunction with the Second International Conference on Principles and Practice of Constraint Programming (CP96)*, 1996.

[123] Peter Van Roy, Michael Mehl, and Ralf Scheidhauer. Integrating efficient records into concurrent constraint programming. In *International Symposium on Programming Languages, Implementations, Logics, and Programs*, pages 438–453, Aachen, Germany, September 1996. Springer-Verlag.

[124] Ralf Scheidhauer. *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz*. PhD thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, December 1998. in German.

[125] Joachim Schimpf. Personal communication, April 2001.

[126] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, 8-11 July 1997. The MIT Press, Cambridge.

[127] Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, Schloss Hagenberg, Linz, Austria, October 1997. Springer-Verlag, Berlin-Heidelberg.

[128] Christian Schulte. *Programming Constraint Services*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2000. To appear in Lecture Notes in Artificial Intelligence, Springer-Verlag.

[129] Christian Schulte. *Oz Explorer - Visual Constraint Programming Support*. The Mozart Consortium – Mozart Oz 1.2.0, May 2001. Available at `http://www.mozart-oz.org/documentation/explorer/index.html`.

[130] Christian Schulte and Gert Smolka. Encapsulated search in higher-order concurrent constraint programming. In Maurice Bruynooghe, editor, *Logic Programming: Proceedings of the 1994 International Symposium*, pages 505–520, Ithaca, NY, USA, November 1994. The MIT Press, Cambridge, MA.

[131] Christian Schulte and Gert Smolka. *Finite Domain Constraint Programming in Oz – A Tutorial*. The Mozart Consortium – Mozart Oz 1.1.1, February 2000. Available at `http://www.mozart-oz.org/documentation/fdt/index.html`.

[132] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In Alan H. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 134–150, Orcas Island, Washington, USA, May 1994. Springer-Verlag, Berlin-Heidelberg.

[133] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of AAAI-92*, pages 440–446, San Jose, CA, July 1992.

[134] Helmut Simonis and Abder Aggoun. Search-tree visualization. In Deransart et al. [37], chapter 7, pages 191–208.

[135] Jeffrey Mark Siskind and David Allen McAllester. Nondeterministic lisp as a substrate for constraint logic programming. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 133–138, Washington, D.C., July 1993.

[136] Jeffrey Mark Siskind and David Allen McAllester. Screamer: A portable efficient implementation of nondeterministic Common Lisp. Technical Report IRCS-93-03, University of Pennsylvania, Institute for Research in Cognitive Science, 1993.

[137] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Current Trends in Computer Science*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, Heidelberg, New York, 1995.

[138] Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.

[139] Frieder Stolzenburg. Membership-constraints and complexity in logic programming with sets. In Franz Baader and Klaus U. Schulz, editors, *Frontiers in Combining Systems*, pages 285–302. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996.

[140] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1997.

[141] Edward Tsang. *Foundations of Constraint Satisfaction*. Computation in Cognitive Science. Academic Press, 1993.

[142] Pascal Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, Cambridge, MA, USA, 1999.

[143] Pascal Van Hentenryck and Yves Deville. The cardinality operator: A new logical connective for constraint logic programming. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 383–403. The MIT Press, Cambridge, MA, Cambridge, MA, USA, 1993.

[144] Pascal Van Hentenryck, Yves Deville, and Choh-Man Ten. A generic arc-consistency algorithm and its specialization. *Artificial Intelligence*, 57:291–321, 1992.

[145] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1–3):139–164, October 1998.

[146] Clifford Walinsky. CLP(Σ^*): Constraint Logic Programming with Regular Sets. In *Proceedings of the International Conference on Logic Programming*, pages 181–190, Lisboa, Portugal, 1989.

[147] Joachim P. Walser. Maximize socializing in golf. `http://www.ps.uni-sb.de/~walser/golf.html`, June 1998.

[148] David Waltz. Understanding line drawings of scenes with shadows. In Patrick Henry Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw Hill, 1975.

[149] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Artificial Intelligence Center, Menlo Park, CA, USA, October 1983.

[150] Jörg Würtz. *Lösen kombinatorischer Probleme mit Constraintprogrammierung in Oz*. PhD thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, January 1998. In German.

[151] Neng-Fa Zhou. A high-level intermediate language and the algorithms for compiling finite-domain constraints. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 70–84, Manchester, UK, June 1998. The MIT Press, Cambridge.

Index

Symbols

D

E

F

O

P

R

S

T

V

W